Automating and Simplifying Agreement and Secrecy Verification using PVS

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science in Computer Science
in the
University of Canterbury
by
André Renaud

University of Canterbury
2001

I would like to thank my supervisor, Paddy Krishnan, of his help with this work, especially with the required motivation, and Jane Mckenzie for her help with the writing style.
I would also like to thank my family, as well as my partner Patti, for their continued support, and reminders that I should be working harder.
Finally, I would like to thank all of the students who I have studied alongside: Tim, Mike, Carl, Andreas, Theuns, and Pramudi.

Abstract

In this thesis we present a system for assisting with theorem proving of security protocols. The desirability of theorem proving is examined and a method of automating the encoding, and some sections of the proof, are demonstrated. We also discuss various aspects of two different classes of security properties: secrecy and agreement. We demonstrate how our system can be used via two case study protocols, NetBill and SET. The proof can be decomposed into various sub-lemmas, most of which can be proven automatically, and then used to simplify the proofs of the final theorems of interest.

Table of Contents

List of Figures

Listings

Acknowledgements

I would like to acknowledge Jon Millen of SRI for providing us with the PVS sources to his encoding of the FFGG[Mil99] protocol. This was vital to our own encoding style. I would also like to thank Jorge Cuellar of Siemens Research, Munich, for his various comments and encouragement.

Chapter I

Introduction

The growth of the Internet has led to an increase in the desire for secure electronic transactions. The security of these transactions is generally not something that can be easily determined. If a flaw is discovered in an installed system, it is both hard and expensive to fix and also dangerous to the security of the overall system. Because of this it is better to attempt to ensure that the protocol is error free before it is installed.

The difficulty in this lies in the complexity of the protocols, and also the subtlety of the errors. The Needham/Schroeder protocol [NS78] was long thought to be the easiest example of a public key cryptography handshake. However it was eventually shown to have a simple flaw which allowed a third party to intercept the communication and then capture and decode any data transfer between the two hosts. This flaw was found by Gavin Lowe [Low96] using the model checker FDR. His use of an automated system started a new interest in the use of formal methods for the assistance of verifying security protocols. Further discussion of this work can be found in Chapter 2.

When developing a security protocol, there can be a large number of exploits relating to practical *implementation* limitations. These are often seen in the form of exploits such as buffer overruns, and are not able to be addressed in the system we will present here. The model we have developed is intended to be used in the context of the design of the protocols. Implementation issues, which are a separate field of research, are not discussed here.

1.1 Motivation

Security protocols generally contain complex dependencies and can quickly become too large to be dealt with in a manual system. We believe that formal methods are an excellent choice for modelling networking protocols, since they provide us with a rigorous framework, and security protocols are well defined. Various techniques have attempted to meet the goal of understanding security protocols. These techniques will be reviewed in Chapter 2. As our literature review will show, there are two main techniques for formal verification of security, namely model checking and theorem proving. Although both have been shown to be capable of analysing protocols successfully, neither are routinely used in an industry setting, mostly due to the high overhead involved. Theorem proving, in particular, is generally only found in limited contexts, since it is a non-automated system, requiring all actions to be provided by the operator. Model checking, in general, is found in some areas of industry, such as hardware development, but is still uncommon in protocol design. Another aspect which has caused problems with the widespread adoption of these formal systems is their difficult interaction with the user. Most formal systems require the protocol to be encoded in the language of the tool which will be used. This is normally a generic system, not specific to security protocols, and, as such, the protocol descriptions are often quite low-level, involving details of which the operator may not be aware. This limits them to the realm of experts. As such, we need a layer which allows users to design protocols in a standard language. We have addressed this problem by performing an automated translation from protocol description to formal model.

1.1.1 Model Checking vs. Theorem Proving

Model checking is the technique for applying some kind of temporal formula to a model of a system. This is a fully automated process. If the system satisfies the formula, then the model checker simply outputs a result in the affirmative, and that is all of the information the user is given. If the system does not satisfy the formula for some reason, then a run of the system in

which the formula is invalid is generated. This run can then be used as the basis for corrections to either the model or the formula. Model checking works essentially by a complete enumeration of the state space in which a given problem is placed. This enumeration thus requires that this space is finite, otherwise complete enumeration would be impossible. Model checking also requires a decision procedure for the system. Thus it asserts restrictions on the logics that can be used, as they must be decidable. Many logics, including first order logic, are not decidable.

Theorem proving is a manual system, and is not automated at all in its most fundamental definition. In general, theorem proving is a framework under which proofs can be carried out, with the tool simply maintaining the state information, and ensuring that the proof does not violate any of its rules. It assists with theorem proving, rather than performing the actual proof itself. Since the proof is actually directed by the user, the full structure for this is available for viewing. Thus if a proof fails to be completed, the user may be able to gain insight through this structure. However, since theorem provers do not normally disprove theorems, a failure for a proof to be complete can either be caused by an invalid theorem, or simply by the user being uncertain as to how to complete the proof. It is only when a proof is completed that the user can be certain about the results. Because of the manual nature of theorem proving, we no longer have the same decidability problems as with model checking; the user must now be able to make the decisions on behalf of the system. Although theorem proving is non-automated, these systems generally have some kind of scripting ability built into them. These are known as strategies.

We have used a theorem proving tool, PVS [COR+95], rather than model checking for several reasons. The finite nature of model checking requires that any given protocol definition be confined to a limited number of runs. Generally with the protocol tools that are written using model checkers, the implementation chooses this number to be one. This one run is usually sufficient to provide information about most of the exploits, as few require information to be carried from round to round. Thus most model checking systems allow only single round attacks to be analysed. However, this single

round limitation can cause problems when dealing with complex protocols.

With a theorem prover, we are no longer bound to a finite state space; we can define any kind of arbitrary state information that we wish. The trade-off is that we no longer have the ability to automatically prove our constraints; we must instead manually guide the proof. This manual guiding is the aspect of protocol verification that we wish to reduce. Initially, before a support system was well established, we estimated the time taken during the proof of a single secrecy theorem of SET to take approximately eighteen hours of human interaction, with approximately four hours of computation time. PVS keeps track of both interaction and CPU time elapsed, but our interaction time includes periods in which the tool was running, but not in use. Our numbers have been adjusted to take this into account. Most of this time is spent redoing the same class of proof repeatedly, as well as recovering from failed proof attempts. A massive amount of mental effort was required to understand the proof, with subsequent difficulties. Due to its vast size, the operator was unable to consider the proof in its entirety, and was instead forced to consider only the current subtree. Due to the highly repetitive nature of the proofs, it was often the case that the technique used to prove one branch would have been forgotten by the time a similar branch was uncovered. This would result in a full analysis being repeated, when the results were already available, but difficult to locate.

1.2 Goals

We intend to demonstrate the technique we have developed to take a realistic protocol description, translate it into a formal description in PVS, and then assist the proof. This is done through the use of both auto-generated lemmas, and hand-coded strategies which work with these lemmas. Our encoding system is discussed, and which lemmas we have chosen to reduce the proof effort. We hope that our system will reduce the amount of effort involved to a level which demonstrates the practicality of using formal methods with respect to protocol design.

Our tool is designed for the semi-expert user, since we believe full-automation is not realistically possible. There will still be interaction with the theorem

proving PVS environment. This interaction is generally simply the instantiation of the appropriate strategy, and as such should not require much more than a list of corresponding strategies to the lemmas on which they work. However, where we have not fully developed strategies, the user in control of PVS, hereafter known as the *operator*, must directly invoke whatever PVS commands are appropriate. This will require a reasonable knowledge of PVS, and at least some knowledge of the protocol involved. The aim is to minimise this as far as possible.

Two papers have been published on two large sections of the work in this thesis. One describing the lemmas and strategies we have used, and how we have developed their auto-generation, is to be published in the New Zealand Journal of Computing 2001 [RK01b]; the other, which covers the environment we have developed, and how the operator interacts with it, is to be presented at the Australian Software Engineering Conference 2001 [RK01a].

1.3 Summary of this Document

Chapter 2 outlines the ground work undertaken in this area, and introduces the necessary technical background of the tools we will employ. In particular, we will discuss how the PVS theorem prover has been used, and how the operator interacts with the environment we have created.

The creation of the translation tool, and the techniques it uses to produce the desired output are described in Chapter 3. We discuss some alternative methods that could have been used, and why we felt our final decisions produced a superior outcome. We will also discuss the generation of assisting lemmas and how we decide which of these should be generated. The application of these lemmas allows us to recognise patterns, which we, in turn, develop into strategies to speed up the interaction with the theorem prover. We will investigate how these strategies are applied, and the completeness of this automation in the various areas of the protocol analysis. This chapter defines the bulk of the original research involved in the creation of the tool.

In Chapter 4 we apply the tool to two different protocols, SET and NetBill, and use the lemmas and strategies generated to prove properties in these case studies. We will show which aspects of our proof tool, and of our proofs,

can be generalised, and which must remain protocol specific. We will also perform a brief effort analysis comparison between our new system and the original hand-coded system. This chapter presents the analyses of our ideas, and the testing of the concepts of this thesis.

Our work is reviewed in Chapter 5, providing a summary of the key aspects. In particular we will focus on how successful our techniques have been with respect to the case studies from Chapter 4, as well as how well we anticipate these results generalising to other protocols, and why we believe this is so. We will also summarise the key lemmas and strategies which provided the greatest improvement in proof time and effort. Finally some aspects of the system that are not yet completed, as well as some other aspects which we have not yet fully studied, but appear to hold some interest, will be discussed.

Chapter II

Preliminaries

2.1 Protocol Verification

The earliest work on formal specification of protocols [BAN89] develops a logic for discussing protocols, and defining properties which should hold. This work does not discuss any form of implementation issues; it is a formal description of the framework required for reasoned discussion about these protocols. This work has been referred to by most of the other work discussed in this chapter, and although we have not used the logic directly, some of the concepts that it defines have been adapted into some of the works which we have used.

2.1.1 Model Checking

One of the earliest formal protocol analyses [Low95] was performed on the Needham-Schroeder protocol [NS78]. This work demonstrated how the Needham-Schroeder protocol could be broken using a man-in-the-middle attack. This work was of interest because the protocol had long been used as the example of how to use public/private key handshaking to set up a shared common key, and its simplicity, three messages, meant it was always believed to be secure. A simple protocol, assumed to be correct for seventeen years, having a flaw, emphasises the need for further analysis of other protocols. This prompted the initial work on formal methods in security protocols.

A system designed primarily to deal with the model checking of security protocols has also been developed [MCJ97], and contains many of the same ideas we use, for example the concept of *closure* described in this work is very similar to our concept of *knowledge*, discussed in Section 3.4.1. This

work also discusses a method of simplifying protocol descriptions. Rather than forcing the user to discuss protocols directly in the language of the model checker, CSP in this case, there is a simplified notation based on the commands *send*, *receive*, and *newnonce*. This language is not discussed in depth, but the ability to perform this translation automatically, as we have, is mentioned. Casper is another protocol model checking system, which is further discussed in Section 2.1.3.

SET has also been analysed in [MS98], and one of the authors has built an interesting tool to automate some of the procedures, [Mea96]. This tool, the NRL Protocol Analyser, (named after the Navy Research Laboratory), is not publicly available. Thus, we have not been able to review it in any realistic way. An overview of the literature of this tool shows that it is a Prolog based system, and similar in many ways to the model checking systems discussed in [MCJ97]. For this reason we believe it will still suffer from the same problems as other model checking systems, as described in Chapter 1. It also requires the operator to encode the protocol directly in the proof system, thus requiring extra effort on the operator's behalf.

2.1.2 Theorem Proving

One of the pioneering works on theorem proving of security protocols [Pau99] involved the verification of TLS. In this work the inductive technique is used to assist in the analysis. This technique is quite simple: the proof is split into two halves. The first half states that a base case is safe. This is generally trivially proven, as the base case is often an empty run. The second half states that, given a valid run of the protocol in which our security property holds, no valid extension of this run will break our security property. This is very similar to the standard mathematical induction system. It is also noted that they required over two hundred lemmas for completion of the final results. In this work, the theorem prover Isabelle [Pau01] was used. This tool requires a much deeper embedding than PVS, and would thus require a greater number of sub-lemmas. Portions of SET have also been analysed using a similar technique [BMPT00]. This analysis, while on a different portion of SET to our system, has allowed us to gain a greater understanding of the SET

protocol, as well as highlighting some additional features for analysis which we would otherwise not have discovered.

A PVS encoding for describing a necessarily parallel attack was described in [Mil99]. In this work the fictitious FFGG protocol is analysed, which requires an arbitrary number of parallel spies to discover the flaw. Another paper on protocol independent secrecy [MR00], also developed by the same group, expands on the concepts of the parallel attack system. This work also discusses the details of the logic behind their PVS implementation in greater depth. This work was fundamental to our own encoding.

The field of theorem proving and security protocols is quite recent, compared to that of model checking. Its non-automated nature has prevented its widespread use.

2.1.3 CAPSL and Casper

CAPSL [DM99], the Common Authentication Protocol Specification Language, is a protocol description language developed at SRI, under a DARPA funded project. A tool called Casper [Low98] was developed to automate analysis using a model checking system. It converts from a protocol description to the process algebra CSP [Hoa85], which can then be checked in the model checker FDR [Ros94]. Casper contains many similarities to CAPSL, especially in the message description language. Most of these similarities stem from the common standard which most protocol designers use. Casper, however, has one difference; it is also a proof tool assistant. It attempts to prove the properties which it is given, whereas CAPSL is a system for specifying these, but does not fully describe a system under which they can be proven. CAPSL does specify an intermediary language, called CIL (the CAPSL Intermediate Language), and a method for translating into this language, but it is not a proof environment, and lacks support for actually performing any analysis. A comparison of the two systems is given in [Low98]. Casper uses a scenario based system, in which the details of the analysis are guided by a scenario specification. This specification can contain constraints on attributes such as the number of runs, or hosts involved. The results of the analysis are thus only valid to the scenario specified.

2.2 PVS

2.2.1 Details

PVS [COR+95] is a theorem proving tool developed by SRI. It is a generalised tool, providing an interactive environment suitable for mathematical and theoretical modelling. An example of the input to PVS is shown in Figure 2.1. This is a simple theory of the factorials of natural numbers. We initially define a predicate, `factorial`, which is quite simple: if X is greater than 0, then factorial(X) is $X \times$ factorial($X - 1$), otherwise it is 1. We also define several theorems, building progressively in interest until our final theorem, `factorial_greater_or_equal`, which states that the factorial of a number is always greater than or equal to the number itself. The proof tree for this simple theorem is shown in Figure 2.3. This is the standard PVS tree diagram used in this thesis when we discuss proofs. The commands entered to perform the proof are the same as those shown in the nodes of the proof tree. The proof tree is useful as it shows the structure of how the proof is conducted. In this example, the first branch indicates the induction over X. Thus the left hand side denotes the case where $X = 0$, while the right hand side denotes the inductive step.

In the recursive definition of *factorial*, PVS requires us to define what is known as a *measure* function, X in our case. These measure functions are used to define a bounding on the recursion, that is, as the measure function decreases, the recursion must near completion.

When proving theorems in PVS, we are shown the current state of the proof in what is known as a *sequent*. The sequent is broken into two parts by a horizontal line. The formulae above the line are known as the *antecedents*, and those below are called the *consequents*. The interpretation is that "the conjunction of the antecedents implies the disjunction of the consequents"[COR+95]. There can be any, including zero, number of terms in either of these two parts. An example of these is shown in Figure 2.2. This is how the formula is displayed in the PVS environment. The formula is read as: factorial(j') $\geq j' \Rightarrow$ factorial($j' + 1$) $\geq j' + 1$

However on occasion we will require greater details of the exact nature of

```
sample: THEORY
 BEGIN

  W, X, Y, Z: VAR nat

  factorial(X): RECURSIVE nat = IF (X > 0) THEN
        X × factorial(X − 1) ELSE 1 ENDIF
     MEASURE (X)

  greater_equal_implies_greater_equal: THEOREM
    ∀ W, X, Y, Z: W ≥ X ∧ Y ≥ Z ⇒ W + Y ≥ X + Z

  factorial_greater_zero: THEOREM ∀ X: factorial(X) ≥ 1

  multiply_positive: THEOREM ∀ X, Y: X ≥ 1 ∧ Y ≥ 1 ⇒ X × Y ≥ 1

  multiply_by_1_greater: THEOREM
    ∀ X, Y: X ≥ 1 ∧ Y ≥ 1 ⇒ X × Y ≥ Y

  factorial_greater_or_equal: THEOREM ∀ X: factorial(X) ≥ X
 END sample
```

Figure 2.1: Sample PVS Theory

```
factorial_greater_or_equal:

  {-1}  factorial(j') ≥ j'
  ------------------------
  {1}   factorial(j' + 1) ≥ j' + 1
```

Figure 2.2: Sample Sequent from PVS

the PVS encoding. In these cases we will use a small example of PVS code.

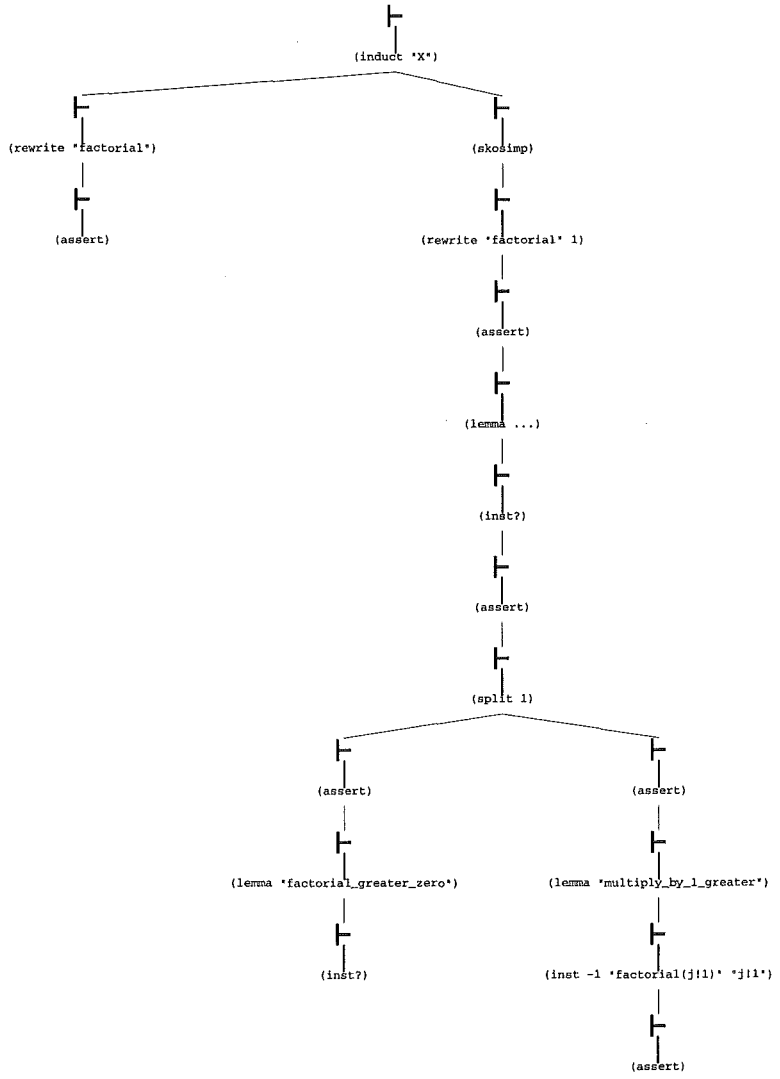


Figure 2.3: Sample PVS Proof Tree

The commands used are as follows:

induct This is used when we wish to use an inductive technique in our proof, with a statement of the form: $\forall\ X : p(X)$. We then generate two sub-lemmas, $(X = 0) \Rightarrow p(X)$, and $p(X) \Rightarrow p(X + 1)$. This is the standard mathematical use of induction, and works automatically on various types, including lists, integer numbers, etc...

rewrite We use `rewrite` when we wish to replace one of our predicate names with the actual definition of the predicate. Thus, if we have an expression of the form $(X = 0) \Rightarrow p(X)$, which is unprovable without knowing the details of p, we would use `rewrite` to reveal these details. It simply expands the body of the definition of the function, p in this case, into where that function is used.

assert This is shorthand for a combination of various other PVS commands. However for our purposes we normally use it for boolean simplification. It applies standard decision procedures. For example, if we had the expression: $(X \Rightarrow Y) \wedge X$ and we wished to prove Y, `assert` could solve this for us.

skosimp This is shorthand for skolemize and simplify. The simplifications are a simple flattening (see below) of conjunctive expressions in the antecedent.

lemma Here we instantiate some previously defined lemma. PVS does not force lemmas to be proven before they are used, but it does prevent forward referencing of lemmas, which prevents circular proofs from being constructed. The normal method is to write the lemmas starting with the simplest, and move down in order of difficulty. Proofs can then be performed top down in the file. This is the method we have used for our encoding.

inst We use inst to give values to expressions such as $\forall$ (in the antecedent), and $\exists$ (in the consequent). There is an alternative version of this command, **inst?** which attempts to automatically guess the appropriate instantiations. This is useful in small scenarios where there are few candidates, but in larger proofs, where there are a large number of possible answers, PVS does not often guess correctly. There is also a version of **inst?** which can attempt to perform a full evaluation in which it will produce multiple possible instantiations. It is written in PVS as **inst? :if-match all**. This is useful for finalising proofs, where

there are few arguments, but it can rapidly increase to a large number of cases. For example if the expression: $\exists A, B : p(A) => p(B)$ required instantiating, and there were four candidate values for both A and B, we would now have sixteen instantiated expressions.

split This will split the proof into two subproofs based on either an $\vee$ (in the antecedent), or an $\wedge$ (in the consequent).

Other commands used in this thesis, but not shown in our example are:

flatten This simply converts expressions of the form $A \wedge B$ (in the antecedent), and expressions of the form $A \vee B$ (in the consequent) into two separate expressions.

then This allows us to join several commands together into one. This is used when calling PVS functions which expect to take only one argument, such as the strategy declaration, `defstep`.

repeat This will continue to execute its argument, another PVS method, until it has no effect on the current sequent. There is an alternative version called **repeat***, which will run itself recursively on any child nodes that are generated, i.e.: when **split** is called.

2.2.2 Strategies

PVS strategies are simply collections of PVS commands, which are then given a name. Using our factorial example, we could remove the body of the final right hand branch by using a strategy such as that shown in Figure 2.4.

In this example we can see that the four commands which make up the final branch could now be replaced with a single command, `(factorial_strategy)`. The final two arguments of the `defstep` command are strings which describe its functionality. The first will be shown in the tree generated; the second will be shown if verbose options are enabled. Normally we would like our strategies to be as general as possible, so lines such as `(inst -1 "factorial(j!1)" "j!1")` are undesirable, as they contain too much specific information, the consequent number, -1, and the exact variable

```
(defstep factorial_strategy ()
    (then
        (assert)
        (lemma "multiply_by_1_greater")
        (inst -1 "factorial(j!1)" "j!1")
        (assert))
"factorial_strategy" "removes the final branch of the facorial proof")
```

Figure 2.4: Sample PVS Strategy

names, `j!1`. We can often avoid these by passing any specific values in as arguments, or in the case of `inst` we can often use the alternatives `inst?` or `inst? :if-match all`.

2.2.3 Inline Notation

In this thesis we use, where convenient, the standard mathematical theorem notation, rather than the exact PVS code, to represent our axioms, lemmas, and theorems. This is simply for readability, since PVS is a normal text based system so it has little symbolic notation; most mathematical terms are spelled out in full text. Thus our example code is not directly loadable into PVS, although it does represent the same concepts. We also distinguish between predicates and theorems through the following scheme: predicates take some arguments, and are thus represented as:

$$\text{greaterThan}(A, B) : A > B$$

whereas lemmas are simply statements of truth, and will be represented as:

"5 is greater than 3" Lemma: $5 > 3$

2.3 Bringing it all together

While we have not taken all of the semantics relating to multiple spies used in [Mil99], we have used the initial encoding as the basis for our initial model. Even though it has since evolved in quite a different direction, the encoding

style used in this work is definitely still visible in our system.

We have used sections of the work in [MCJ97] for the development of our input language. However, the discussion of the input language in that paper is quite slim, as the work emphasises the logic used to specify the protocol properties. We have not used a full formal logic for our security properties, as we desire our input description to be as simple for the protocol designer to read as possible.

We have modelled our input language heavily on the basic syntax of CAPSL. We have simplified some of the syntax slightly, such as encryption where we do not require the explicit naming of the encryption form, e.g. public/private and shared, as this can be determined from the key involved. We have also extended the input language slightly, adding some process algebraic semantics to allow for more dynamic protocols to be described. We have used the CCS [Mil90] process style as the basis for these extensions. A complete definition of our language is discussed later in Chapter 3. CAPSL contains considerable information outside the message flow description. This information has not been used in our model. This is mostly due to the specifics of our implementation, and the limits that we have imposed. We have instead modified these sections to focus on the data in which we are interested. For example, computational rules have been omitted, and security properties have been simplified, so that the effort by the operator is minimal. Our input syntax also shares many features with Casper, particularly with the security property specification, of both `Secret` and `Agreement` which we have mirrored directly.

Chapter III

Translator

The first aspect of the research involved in this project is the construction of a translation tool. The input language we accept and the techniques used to simplify the user interaction with the theorem prover are the primary contributions we have made towards the goal of easing protocol proofs. In this chapter we will discuss the output from the tool, and how the various ideas are combined in the final result to produce a simple, fast and usable system.

In an effort to remedy the problems relating to the manual nature of theorem proving, we have developed a system for breaking down the proofs into manageable chunks and a selection of strategies for automatically proving some of these chunks. This has reduced the time taken for a secrecy proof to approximately one hour of interaction, with only about fifteen minutes of computation time. We have also introduced another benefit: since the proofs are broken down, we can now reuse various aspects of these within all proofs, and thus subsequent proofs take much less time, since we do not need to reprove anything we have already done.

3.1 General Ideas

The translator is designed to transform the input protocol definition into a suitable PVS encoding and any auxiliary subsystems we might require to usefully manipulate this encoding. It uses a Bison [DS95] based parser to convert the input into a parse tree, and then convert this parse tree into the required subsystem. There are two main sections of the parser, as with most parsers: parse tree construction and parse tree inspection. We will not discuss the construction of the parse tree, and simply state that it is

constructed by using the standard LALR [ASU86] parsing technique.

The focus of this translation tool is to simplify the effort of manually transforming the protocol specification into a formal model, and to assist in the manipulation and proving of properties of this formal model. The development process for this tool has been incremental, with a hand-coded model being gradually constructed to meet the desired requirements. This hand-coded system is then gradually generalised into the auto-generation tool. The input to our translation tool is similar to that of CAPSL and Casper, as discussed in Chapter 2. Most of the basic message concepts remain the same; we have only modified the auxiliary structure, as well as making small changes to the methods for describing the overall message flow.

The primary goal has been to simplify verification of the proof. To this end we have developed a series of additional supporting features to assist the user. We hope to provide near automation, since, with a theorem proving environment, full automation is impossible in the general case. The simple aspects of the proofs have been automated, and strategies and supporting lemmas have been provided to divide the large and complex sections into manageable components.

The two security aspects which we have chosen for our analysis are secrecy and agreement. Secrecy denotes the idea of a given token being secure, and unable to be learned by a spying third party. Secrecy is a global invariant, similar to safety properties in standard formal methods, and thus has an easily defined specification. Its ease of specification, and strict rules, makes it easier to automate under our system. Agreement is the idea that a specific token is eventually known to two or more hosts. Agreement is thus an eventuality, or a liveness property. This makes the specification of agreement more involved than that for secrecy, since we cannot globally qualify it, as it is only true in some states of the system. As with liveness properties in general, it is hard to prove.

3.2 Translation Tool Syntax

We have adapted ideas from all the systems discussed in Chapter 2 to create our own protocol specification system. We have taken the base syntax

from CAPSL and Casper, but included some of the CCS syntax, and ideas on representation and structure from both Message Sequence Charts and Graphs.

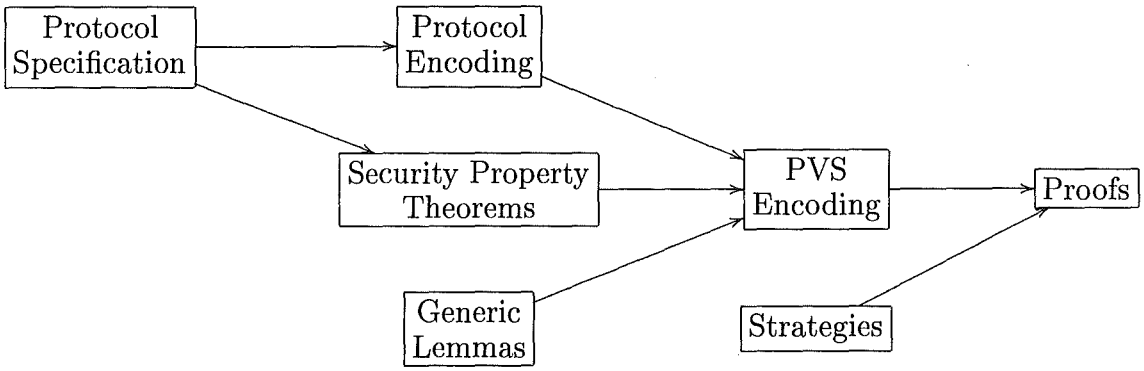


Figure 3.1: Translator Process Pipeline

The translation tool we have developed follows a simple basic model, as shown in Figure 3.1. We take the protocol specification from the user, and extract two subsystems: the protocol encoding and the security property theorems. We then use our protocol independent lemmas along with these two to form our overall PVS encoding. Finally, together with the strategies we have developed, we can output the final proofs. We have attempted to automate each step in this pipeline, although some are far more difficult than others. This section will now discuss the format of the input language we accept, with Section 3.2 describing the semantics associated with this language.

We have based the core protocol definition portion of our language on the CAPSL and Casper languages, with some additional concepts from other systems, as discussed in Section 2.3. However, our auxiliary definitions have been designed separately to include features in which we were interested. This difference is not particularly important, since the protocol description is the key issue. The auxiliary information simply provides hints to the translator. Thus, the protocol description is split into two logical sections: the prologue and the protocol definition. The prologue contains descriptions

of the variables involved, the properties we wish to hold true, the initial knowledge of the hosts (including the Spy), and what capabilities we want the Spy to have. Following the prologue is the protocol and message definition, where we have followed the CAPSL/Casper syntax. This consists of two types of messages: either those which describe the structure of the protocol, or those that describe a literal communication between two hosts. These two message types are described in Section 3.3.1.

A BNF grammar for the language we have designed is defined in Figure 3.2.

The process element of the state is defined using a similar syntax to those found in [Mil90]. The three operators we have taken from this system can be combined using the following simple grammar:

$$
\begin{aligned}
\text{Protocol} \rightarrow\ & \texttt{X} \\
\mid\ & \text{Protocol} \mid \text{Protocol} \\
\mid\ & \text{Protocol} + \text{Protocol} \\
\mid\ & \text{Protocol} \ ; \ \text{Protocol}
\end{aligned}
$$

This defines the following operators, which we have also used in our description:

- | : Parallelism, two processes run in parallel. There is no fairness restriction here. We denote this with the term `parallel` in our encoding.

- + : Choice, the arbitrary decision between two possible processes. Denoted by `choice` in our encoding.

- ; : Sequencing, simple concatenation of one process after another. Denoted by `sequence` in our encoding. See Figure 3.6.

- X : An atomic action. In our encoding this corresponds to a message sent from one host to another. These become the basic communications of our protocol.

Full Protocol	→	`Protocol` <String>
		<Constants>
		<Variables>
		<Initials>
		<Spy>
		<Specifications>
		<Protocol>
Constants	→	`Constant` <Declaration> { , <Declaration> }
Variables	→	`Variables` <Declaration> { , <Declaration> }
Initials	→	`Initial` <Knowledge> { , <Knowledge> }
Knowledge	→	`knows` (<Host>) = <Variable> { , <Variable> }
Spy	→	`Spy` <SpyType> { , <SpyType> }
SpyType	→	`Observation`
	→	`Synthesis`
	→	`Destruction`
Specifications	→	`Spec` <SpecType> { , <SpecType> }
SpecType	→	`Agreement` (<Host> , <Host> , '[' <Variable> { , <Variable> } ']')
	→	`Secret` (<Host> , '[' <Variable> { , <Variable> } ']')
Declaration	→	<Variable> { , <Variable> } : <VariableType>
VariableType	→	`Host`
	→	`Nonce`
	→	`Key`
	→	`Signature`
	→	`Natural`
	→	`Hash`
	→	`Certificate`
Protocol	→	<Message> { , <Message> }
Message	→	<Variable> = <Host> -> <Host> : <Field> { , <Field> }
	→	<Variable> = <Variable> { ; <Variable> }
	→	<Variable> = <Variable> { | <Variable> }
	→	<Variable> = <Variable> { + <Variable> }
Field	→	<Variable>
	→	'{' <Field> { , <Field> } '}' <Variable>
	→	<Variable> '{' <Field> { , <Field> } '}'

Figure 3.2: Input Language Grammar

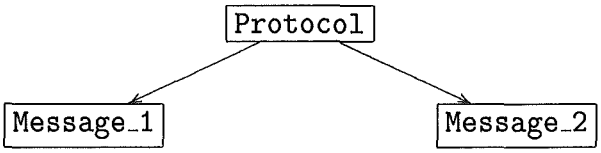


Figure 3.3: Internal Process Tree Representation

This encoding breaks our protocol down into a fairly simple graph structure, where links denote parentage. Thus we could say that there is a link from node `Protocol` to node `Message_1`, see Figure 3.3. For most protocols we can be even stricter than this as most adhere to a tree structure, where the leaf nodes indicate the communication messages, and all those above indicate process messages. The root of the tree is now the controlling process, and all leaf nodes are now communication messages.

The atomic action, X, in our above grammar, is now a message sent from one host to another. It is defined in the following form:

$$\text{Host}_1 \rightarrow \text{Host}_2 : \text{Payload}$$

3.2.1 Message Field Types

As can be seen in our context free grammar, Figure 3.2, there are seven different field types which can appear within messages. There is also one implicit field type, encryption, which we will also discuss here. Each of these types is considered distinct, as will be discussed below.

Host This describes an agent of the system. It can be used to identify target agents, or as the basis for key negotiation.

Nonce A nonce is a special random number in cryptography, which cannot be guessed. Under the assumption of perfect cryptography, we have assumed that no two nonces are ever the same.

Key This is the key to some encryption. We have not made a distinction between public/private and shared keys in the type rules, as we do

this through predicates relating key inverses. That is, for shared keys, the inverse is known if given the original. However this is not true for public/private keys.

Signature This is a cryptographic signature used to verify the authenticity of a field.

Certificate Certificates are often used to verify the identity of a host when given a trusted third party. For example in the SET protocol, hosts use a third party certificate authority from which to lodge and retrieve certificates, allowing individual hosts to directly communicate, but adding the ability to ensure only trusted communication.

Natural We have added the natural type as a general place holder for other data that does not directly contribute to the cryptographic properties of the protocol. Thus it is often used to denote some piece of data which is transferred. For example, if the object of a protocol is to transfer a user's name as a string of characters encrypted with some key, we could not simply drop the name from the protocol, as that would remove meaning, but since we also are not interested in the actual contents of it, we would thus use the *Natural* type to hold this data.

Hash A hash under our system is assumed to be perfect. That is, no two items ever collide, and given a hash it is impossible to guess the fields from which it was constructed. These are known as non-invertible, or one-way, hashes. We do not make a distinction between the different types of hashing; all systems, such as SHA [Bur95], are assumed to be of equivalent strength. Also, due to our higher level of abstraction we do not deal with issues such as hash lengths.

Encryption This is not a strict type, as the other seven here are, as a variable cannot be declared of this type. However, it is used implicitly whenever a key is applied to a field. We use this type in the internal representation of our protocols. We have assumed perfect encryption,

that is, a field can never be guessed given its encrypted text, and no key can be guessed given an example of both the plain and cyphertext.

Each message is then represented as a list of these variables of these various types. When declaring these there is an implicit assumption that they are all unique. For example, if given the following definition:

```
X, Y : Key
A, B : Natural
```

we assume that $X \neq Y$, $A \neq B$. We also assume all types to be distinct, i.e.: $A \neq X$. There are alternatives to this approach, which allow for these items to be possibly equal, but this vastly increases the effort involved in the proofs, and does not offer much benefit for security properties, especially when considering only perfect cryptography.

To demonstrate the various parts of our protocol design, given the protocol defined in Figure 3.4, the state transition system would be similar to that shown in Figure 3.5. The final state, *blank* is further discussed in Section 3.3.2.

$$\begin{aligned} \text{Protocol} &= \text{Message}_1; \text{Message}_2 \\ \text{Message}_1 &= A \rightarrow B : A, \{X\}_{K_1} \\ \text{Message}_2 &= B \rightarrow A : B, \{X\}_{K_2} \end{aligned}$$

Figure 3.4: Simple Two Message Protocol

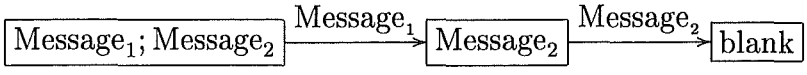


Figure 3.5: Two Message State Transition System

3.3 Translation System Semantics

3.3.1 State Transition System

The overall design of the encoding is as a state transition system. Each state in our transition system is internally represented as a pair, a combination of a process, and a history of previously seen messages.

From this definition, our translator builds the state transition system, as well as a variety of predicates to manipulate it. These predicates are mostly protocol independent, and allow us to easily manipulate the transition system in such a way that it remains consistent, and retains all information we consider important.

We break our entire system down as follows:

- A run of the entire system is called a trace.
- A trace is a list of transitions.
- Each transition is represented as a start state, a message, and an end state.
- States contain a pair, message and history.
- Messages are defined as a source host, destination host, and an associated field.
- Fields are broken down as some combination of trivial tokens, encryption, hashing, signatures and certificates.
- State history is a simple list of messages. This information is maintained in the state for simplicity, but is also available by walking backwards through the trace.

We break down messages into two classes: *processes* and *communications*. A *process* is any message from the input which defines the structure of the protocol, such as `Process` from Figure 3.4. Anything involving any

of the operators ;, | or + is thus a process. A *communication* is any message which defines something literally sent between two hosts. The messages stored in state history will always be communications, as they indicate what has been transmitted between the hosts. Using the sample protocol defined in Figure 3.4, we demonstrate the encoding of each element of this protocol in Figure 3.6. The messages `Message_1` and `Message_2` are thus both communications, whereas `Protocol` is a process.

$$
\begin{aligned}
&\text{isProtocol}(\text{process}_r) : \\
&\qquad \exists\ \text{process}_1, \text{process}_2 : \text{Process} \\
&\qquad\qquad \text{isMessage}_1(\text{process}_1) \wedge \\
&\qquad\qquad \text{isMessage}_2(\text{process}_2) \wedge \\
&\qquad\qquad \text{process}_r = \text{Sequence}(\text{process}_1, \text{process}_2)
\end{aligned}
$$

$$
\begin{aligned}
&\text{isMessage}_1(\text{process}_r) : \\
&\qquad \text{process}_r = \text{said}(A, B, \{A, \text{enc}(K_1, X)\})
\end{aligned}
$$

$$
\begin{aligned}
&\text{isMessage}_2(\text{process}_r) : \\
&\qquad \text{process}_r = \text{said}(B, A, \{B, \text{enc}(K_2, X)\})
\end{aligned}
$$

Figure 3.6: Sample Message Encoding

As previously shown in Figure 3.3, we can decompose our protocol into a hierarchy of messages. Once we have broken our protocol down, we can now work with it in a much more manageable state. However, we are normally only interested in the topmost definition, `isProtocol` in this case, as it is the root of the tree, and thus the process which defines the entire protocol. With this pseudo-tree structure, we can quite easily break down our proofs, i.e. where we would like to have a universally quantified predicate supposition over the entire protocol, we can now break that down into n universally quantified proofs, one for each distinct communication message. Using these proofs, we can then walk up the tree, proving our supposition at each node via the proofs of its children.

3.3.2 State Transition

We have now described the overall layout of a protocol *trace*. However, we need to be able to build on these traces, extending them as the protocol communication progresses. For this we have the concept of a *move*, defined in Figure 3.7. That is, a transition from one process to another over a certain message. These two processes form half of the state information discussed earlier. For implementation issues we require the idea of a *blank* process, that is a process which cannot perform any messages, or be transformed in any way. This is simply for technical implementation reasons and has no impact on our methods. We use these *blank* messages to describe a terminated process, which is similar to the 0 process in CCS.

$$
\begin{aligned}
&\text{move}_{[\text{process}]}(P_1, P_2, M) : \\
&\qquad (P_1 = M) \;\Rightarrow\; P_2 = \text{Blank} \\
&\qquad (P_1 = P_{1a}|P_{1b}) \;\Rightarrow\; (\text{move}(P_{1a}, P_X, M) \wedge P_2 = P_X|P_{1b}) \vee \\
&\qquad\qquad\qquad\qquad (\text{move}(P_{1b}, P_X, M) \wedge P_2 = P_{1a}|P_X) \\
&\qquad (P_1 = P_{1a} + P_{1b}) \;\Rightarrow\; \text{move}(P_{1a}, P_2, M) \vee \\
&\qquad\qquad\qquad\qquad \text{move}(P_{1b}, P_2, M) \\
&\qquad (P_1 = P_{1a}; P_{1b}) \;\Rightarrow\; (P_{1a} = \mathit{Blank} \wedge \text{move}(P_{1b}, P_2, M)) \vee \\
&\qquad\qquad\qquad\qquad (\text{move}(P_{1a}, P_X, M) \wedge P_2 = P_X; P_{1b})
\end{aligned}
$$

Figure 3.7: Move

We do not actually use `move` directly, but we have built another predicate on top of it which manipulates states and history correctly, as shown in Figure 3.8. We use two separate predicates here, since *states* contain more than just process information; they also contain a history of previously seen messages.

Using the state move predicate we can now easily move through the protocol from the initial process description generated from the protocol definition. One last modification that must be made is in protocol initialisation. We have no restriction on the number of rounds of a protocol that can be

$$\text{move}_{[\text{state}]}(S_1, S_2, M) :$$
$$\text{move}_{[\text{process}]}(S_1[\text{process}], S_2[\text{process}], M) \wedge$$
$$S_2[\text{history}] = S_1[\text{history}] \cup M$$

Figure 3.8: State Move

run, and thus have added the ability for protocols to be started at any point throughout a trace. Therefore our trace begins with the root process for the given protocol, from which we can either continue to run as a single protocol, or add new runs as needed. We are allowed to start another parallel run of the protocol at any time. This ensures that exposures of information in one round will not affect any subsequent protocol run. This allows us to model multi-round attacks.

We can demonstrate this using the protocol shown in Figure 3.9. This is a simple linear two-message protocol. Initially we would begin with the root process for the protocol, $P_1; P_2$. From here we choose to perform the P_1 move, leaving only the P_2 message. Now, we instantiate a second round of the protocol, and shift on its first message; P_1. It should be noted that we never see the entire process for subsequent rounds, since we immediately shift on their first message. Thus our second process immediately becomes P_2, as the initial P_1 was consumed in the move. We now have two protocols running in parallel, each with only the P_2 message remaining. We shift on the second instance's P_2 message, and then on the first's. This brings us to a *blank* state, although we could begin a new run. We could continue to create additional runs indefinitely. Thus, there is no limit on how many runs a given protocol will execute, allowing the Spy to model multi-round attacks. An example of a multi-round run is shown in Figure 3.10.

We can describe this in more concrete terms. If hosts A_1 and B_1 are engaged in a run of the protocol, there is nothing to stop a second pair of hosts, A_2 and B_2, from communicating at the same time. These two communications are completely unrelated, and since there are no timing constraints in our system, they run at independent speeds. There is also no restriction

$$
\begin{aligned}
P &= P_1; P_2 \\
P_1 &= A \rightarrow B : M_1 \\
P_2 &= B \rightarrow A : M_2
\end{aligned}
$$

Figure 3.9: Multi-round Protocol Sample

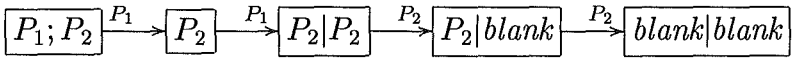


Figure 3.10: Multi-round Protocol Run

on starting time. Thus the second protocol run may begin at the same time as the first, after the first has completed its run, or at any state in between. Further, there is also no restriction on the hosts involved, so A_1 and A_2 may be the same host, and thus knowledge uncovered from one protocol run would be relevant to attacks on the other. However, since we assume that only the Spy is malicious, we are not generally interested in what knowledge non-Spy hosts maintain between rounds.

3.3.3 Encoding of Variables

A problem with this method is the instantiation of the variables in the encoding. If we declare variables directly, in such a way that we have a variable in our encoding for each one declared in our input definition, then they can never change between protocol runs, thus giving the false impression that anything revealed in one transaction remains the same for all subsequent transactions. However, if we simply declare them as names, then it is impossible to force them to remain constant within a given transaction without passing them around to every predicate and lemma which needs their values. To compensate for this, variables are encoded as a mapping from hosts to their respective types. Thus each variable is bound to a host in the protocol. Under most conditions, this binding is relatively easy to determine, and relates well to the instinctive understanding of the protocol. Thus, for

example, public/private keys are bound to their appropriate host, etc...

3.3.4 Protocol Independent Subsystem

There are a number of fixed definitions that are applicable to every protocol generated. These include the data type definitions, as well as the lemmas associated with these data types relating to their internal consistency. We also define all of the fundamental predicates for working with these data types, such as `reveals`, as protocol independent. After the data type definitions, the majority of the protocol independent system definition consists of lemmas relating `parts` with `reveals`. An example of this is shown in Figure 3.11.

$$\begin{aligned}&\text{“Y in parts(T) via X” Lemma :}\\ &\qquad\qquad \text{reveals}(X, Y) \wedge\\ &\qquad\qquad X \in \text{parts}(T) \Rightarrow\\ &\qquad\qquad\qquad Y \in \text{parts}(T)\end{aligned}$$

Figure 3.11: Assisting Simplification Lemma

There are approximately fifty lemmas of this type, which are used to simplify the proof effort. This is done by substituting intuitive ideas with lemmas in order to avoid unnecessarily overloading the user with information. An example of these forms of lemmas is given below. It states that all revealed tokens must also be a member of the *parts* of the trace.

$$\forall\, X, T : \text{reveals}(T, X) \Rightarrow X \in \text{parts}(T)$$

3.4 Support for Proving

3.4.1 Transition System Predicates

The simplest of the transition system predicates is `parts`, which defines which symbols are involved in any of our data structures. For example, the value

`X` would be considered to be a member of `parts` of the following message:

$$\text{Host}_1 \rightarrow \text{Host}_2 : X, Y, Z$$

When values appear as the argument of some kind of function, encryption or hashing for example, they can only be a member of `parts` if the function is reversible. Thus, in the following:

$$\text{Host}_1 \rightarrow \text{Host}_2 : \{X\}_F$$

X would not be a member of `parts` if F was a hash function, which we assume to be one-way, but if F were a key, and thus $\{X\}_F$ represents encryption, then X would be in `parts`, since encryption is reversible.

The predicate `parts` thus defines all symbols which are somehow extractable from a field, message or trace. Since hashing is considered to be one-way in our system, a hashed token is unrecoverable. However encryption is readily broken down, since its content is extractable. The predicate `parts` has no notion of knowledge, so it does not attempt to determine whether a key is known; it simply ignores the encryption.

Following `parts`, we have a similar predicate called `reveals`. We use `reveals` to denote information that is revealed in plain text through some communication. Thus `reveals` works very similarly to `parts`, except it does not break down encryption. This is the first step towards analysing protocols for secrecy. Using the following protocol snippet:

$$\text{Host}_1 \rightarrow \text{Host}_2 : X, \{Y\}_K$$

we would say that X is revealed. However, if K is a key, then Y is not revealed. It should be noted that `reveals` does not have to be used on only trivial tokens, thus in the above example we could say that $\{Y\}_K$ was revealed.

Once we have `reveals` and `parts`, we can now build our predicate for knowledge. The predicate `knows`, as defined in Figure 3.12, allows us to walk through a trace, and determine whether a given host can learn a given token.

$$
\begin{aligned}
&\text{knows}(H, T, X) : \\
&\qquad X \in \text{parts}(T) \wedge \\
&\qquad (\text{reveals}(T, X) \vee \\
&\qquad (\exists\, Y : \text{Field}, K : \text{Key} : \\
&\qquad\quad \text{reveals}(Y, X) \wedge \\
&\qquad\quad Y \in \text{parts}(T) \wedge \\
&\qquad\quad \text{knows}(H, T, Y) \wedge \\
&\qquad\quad \text{knows}(H, T, K)))
\end{aligned}
$$

Figure 3.12: The Recursive Knows Predicate

It attempts to break down encryption, but it will only do so if the key to that encryption can be learnt in some way, either through some other message, or via initial knowledge. This concept of `knows` is based on Millen's encoding of *analz* [MR00].

This definition breaks down as follows. Initially, we require that the symbol in which we are interested, X, appears at some point in the trace, in some readable way, $X \in \text{parts}(T)$. From here, there are two possible ways it could be available to a host, either directly through plain text, $\text{reveals}(T, X)$, or indirectly through an encrypted message, $\exists\, Y : \text{field}, K : \text{Key} \ldots$. However, this encryption may, in turn, contain an encrypted message which must be broken, and so on. This continual nesting of encryption is represented through the use of recursion in `knows`. We can now break down an arbitrary depth of encryption. Unfortunately this recursion is initially unbounded, since we do not know in advance how many levels we will have to search. This is where the predicate `parts` becomes important. We will eventually reach some state where we have n levels of encryption, i.e.

$$Y = \{\{\{\{X_{K_1}\}_{K_2}\}_{\ldots}\}_{K_n}\}$$

However, for any given protocol there are a finite number of message types, and associated with each of them is a finite depth of encryption. Thus

each protocol will have some maximal n for which there cannot possibly exist any deeper encryption. This deeper level of encryption cannot therefore be covered by `parts`, and the first aspect of `knows` will now fail. This is how we prevent `knows` from perpetually attempting to increase the encryption depth.

3.4.2 Initial Knowledge

We encode the initial knowledge of the system as axioms relating to hosts. Thus for any given host, there exists an axiom which ensures that the `knows` predicate is satisfied for its initial knowledge. These axioms are defined directly from the `Initial` section of the protocol definition. For example, given the specification shown in Figure 3.13, the axioms generated would be as shown in Figure 3.14.

```
Initial
        Knows(A) = K_A, K_A', K_P
```

Figure 3.13: Sample Initial Knowledge Specification

"Initial Knowledge for A" Axiom :

$\forall\, T$: Trace :

knows(A, T, K_A)

knows(A, T, K_A')

knows(A, T, K_P)

Figure 3.14: Sample Initial Knowledge Axioms

3.5 Secrecy Lemma Generation

Secrecy lemmas can be broken into two broad categories: message based, and protocol based. Message based lemmas relate to a single message and do not require any information about their position in the protocol as a whole.

Protocol based lemmas do not deal with individual messages at all, but rather how these messages relate to form the overall system.

There are two sets of lemmas generated for message based secrecy. The first set, key secrecy, is quite general, and does not need to be regenerated for each individual secrecy theorem. The second set, token secrecy, relates highly to the secrecy lemma for which they were defined. This second set thus appears once for every secrecy lemma defined, whereas the first set will appear only once per protocol.

3.5.1 Key Secrecy

It should be noted that our current system assumes that no key will ever be revealed in plain text, as this would render such a key useless for encryption purposes. This assumption could cause problems were any protocol to reveal any of its keys, and in such a scenario it would be necessary to decompose the key secrecy lemmas further, so that rather than dealing with all keys as a whole, this protocol deals with them individually, ignoring any keys which were revealed in plain text. This would not severely influence any of the later results, but would increase the effort involved in proving the initial lemmas. The key secrecy lemmas are of the form shown in Figure 3.15. These are fairly easily proven, since they deal only with the revealing of the key in plain text.

"M does not reveal any key" Lemma :

$$\forall\ K : \text{Key} : \neg\text{reveals}(M, K)$$

Figure 3.15: Key Secrecy Lemmas

Token Secrecy

Token secrecy is very similar to that for keys, except we can now no longer generalise it for an entire class of variables. This gives us a group of lemmas for each token relating to individual messages, similar to those for key secrecy

shown in Figure 3.15, as well as a final lemma proven using those in this group. An example of this final lemma encoded for a nonce, N, is shown in Figure 3.16. These lemmas would only be generated for those tokens which have been declared secret in the input file.

"No trace reveals N" Lemma :

$$\forall\, T : Trace \;\; : \text{valid}(T) \Rightarrow \neg\text{reveals}(T, N)$$

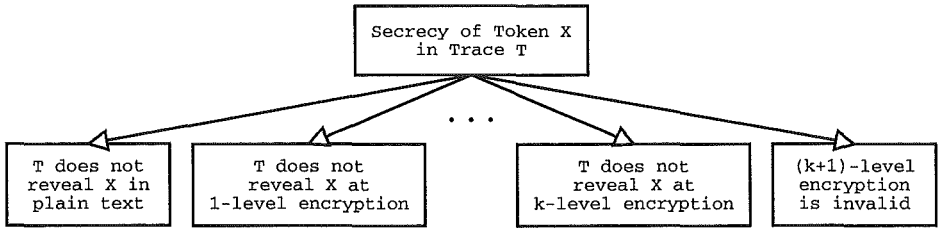
Figure 3.16: Final Token Secrecy Lemma

3.5.2 Global Secrecy Theorem Structure

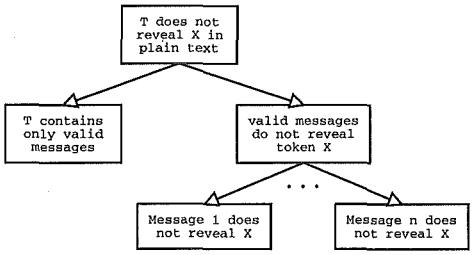
While it is useful to generate all these individual lemmas, what we require is a few high level lemmas which can be used in many scenarios very easily. We thus build a hierarchy of lemmas as shown in Figure 3.17. From this figure we can see that the protocol dependent lemma generation is then abstracted into a protocol independent lemma, i.e. the individual message key secrecy lemmas can be used to prove the general message key secrecy lemma. Since the individual lemmas are protocol dependent, it is hard to generalise their use, as they will change between different protocols. Thus, when we combine them into a generalised lemma, we now have a base on which we can build generic proofs that will work under any protocol, once the individual messages have been used appropriately.

From Figure 3.17, we can see that the proof of the secrecy theorem, as shown in Figure 3.17(a), can be readily broken down into more easily managed parts.

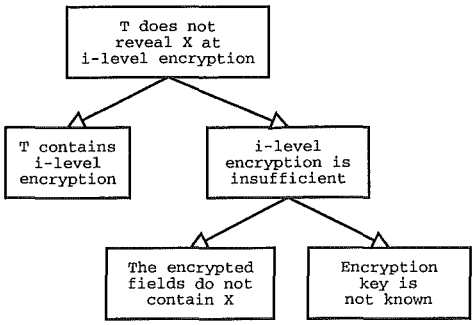
Figure 3.17(b) shows the structure of the proof pertaining to the plain text section of the overall theorem. Here we initially ensure that the protocol contains only valid messages, as this is where we bring the protocol definition of the messages into our system. We then must prove that none of the messages in the protocol reveal the given token in plain text. This is a simple proof, but for simplicity, and for reuse, we break this down into a lemma for each message, as discussed in Section 3.5.1.



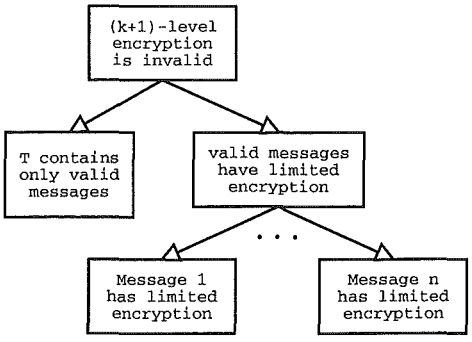
(a) Secrecy Lemma Breakdown



(b) Revealed in Plain Text



(c) Revealed at i-level of encryption



(d) n+1 levels of encryption is invalid

Figure 3.17: Secrecy Theorem Proof Hierarchy

Figure 3.17(c) shows the general case for each level of encryption. For a simple protocol involving only a single level of encryption, there would be one lemma of this type. However, most protocols involve at least two levels of encryption, especially protocols which involve three or more hosts, and, as such, there could be several of these lemmas. These lemmas are the most complex portion of the secrecy proof. They are broken down into two basic parts: finding all fields encrypted at the relevant depth, and then proving that these fields do not violate our secrecy criteria. Finding the field is a simple brute force task, since we can simply instantiate every message in the protocol, examining their contents until we find a list of appropriate candidates. We then take these candidates individually, and prove that they do not violate our criteria. There are two ways this could be proven: either the message does not contain the relevant token, or the key involved in the encryption is safe. Proving it does not contain the relevant token is trivial, and is done automatically. However, proving that the key is safe is difficult. We can isolate this proof off with its own lemma, and prove it separately, in a similar manner, except this time the token in which we are interested is the appropriate key.

Figure 3.17(d) shows the final step in the secrecy theorem proof. Once we have exhausted all the depths of encryption, we need a method to inform the theorem prover that we are in fact completed. We thus have a short final lemma which states that there is no encryption above a certain level in this protocol, so there is no point in attempting to search for it. This proof is quite short, but due to its raw nature, it is not easy to automate in an elegant way (although it is automated in an inelegant way).

Further discussion on our automated techniques for this are discussed in Section 3.7.1.

3.5.3 Protocol Based Secrecy

Protocol based security, as implemented, is currently limited to encryption depth, that is, how many levels of nested encryption there are in a given protocol. This has been discussed earlier in Section 3.4.1.

3.6 Agreement Lemma Generation

The agreement lemmas and predicates are the most complex section of the output encoding. With agreement, we are attempting to prove that there exists *some* run in which the agreement condition holds. Because this is not a universally quantified condition, we must produce a specific run of the protocol in which to test it. Since our translator is relatively simple, it does not have the capacity to properly evaluate which run is applicable in a reasonable amount of time. We have thus chosen the following heuristic technique. We first find a message which contains the given token in any extractable form, i.e. plain text, or encryption, but not hashing. We then backtrack from this message to the beginning of the protocol, selecting the appropriate choice and parallel processes as we go. When we have backtracked to the initial protocol definition, we can reconstruct the list of actual messages we have seen in order to obtain our prospective message. This is the initial extension guess which is encoded in our output. We have chosen to find the first suitable message, rather than the last, so that these extensions are as short as possible, thus reducing the amount of effort required in the proofs. It should be noted that the heuristic can perform rudimentary analysis of the protocol to ensure it does not get confused by protocol complexities such as looping and nesting.

Once our heuristic run has been generated, there are still two additional sets of lemmas which we generate. As in security, we generate a set of token secrecy lemmas, since we generally only attempt agreement on secure tokens. This could cause problems for protocols in which we are attempting agreement on an insecure token, as these lemmas would thus not be applicable. This scenario would limit the usefulness of our theorem break-down method, as we would no longer be able to use these lemmas. However we believe that making the assumption that no agreement token candidate is ever revealed in plain text is reasonable, as this would render the token immediately insecure, and could thus be easily learned via snooping. Once these secrecy lemmas are generated we then generate two sub-lemmas, one for each host involved in the agreement. This is a simple technique for splitting the final proof into manageable chunks. For example, if the final agreement theorem appeared

as shown in Figure 3.18, then we would also generate the two sub-lemmas shown in Figure 3.19, for host A, 3.19(a) and for host B, 3.19(a).

"A and B agree on X" Theorem :

$$\forall\, T_1 : \exists\, T_2 :$$
$$extends(T_2, T_1) \wedge$$
$$(knows(A, X, T_2) \iff knows(B, X, T_2))$$

Figure 3.18: Fundamental Agreement Theorem

"A knows X" Theorem :

$$\forall\, T_1 : \exists\, T_2$$
$$extends(T_2, T_1) \wedge$$
$$knows(A, X, T_2)$$

(a) Agreement assisting lemma for A

"B knows X" Theorem :

$$\forall\, T_1 : \exists\, T_2$$
$$extends(T_2, T_1) \wedge$$
$$knows(B, X, T_2)$$

(b) Agreement assisting lemma for B

Figure 3.19: Agreement Theorem Assisting Lemmas

Unfortunately, however, we cannot use both of these lemmas in our final proof. Since the general theorem, Figure 3.18, states that there exists a trace which is valid for both hosts, and if we were to simply use both of these lemmas there is no guarantee that the traces used in Figures 3.19(a) and 3.19(b) are the same. We can thus only use one of these lemmas in our final proof. Further discussion on this can be found in Section 3.6.3.

3.6.1 Heuristic Example

Using the protocol shown in Figure 3.20, our heuristic extension generation would initially attempt to find the token of interest, `Data`, in a suitable message. In this case `SendData` contains the token. It would then backtrack from here to `Heuristic`, where it would discover that it needs to have

performed the `NegotiateIdentities` process first. It would then unfold `NegotiateIdentities`, discovering that `FirstA`, and `ThenB` are both needed, so they would be prepended to the run. Finally it would discover that it is at the beginning of `Heuristic`, and that this is the root process, so there is no further discovery necessary. This would result in the run being: `FirstA` $\rightarrow$ `ThenB` $\rightarrow$ `SendData`, as we would anticipate.

$$
\begin{aligned}
\text{Heuristic} &= \text{NegotiateIdentities; SendData} \\
\text{NegotiateIdentities} &= \text{FirstA; ThenB} \\
\text{FirstA} &= A \rightarrow B : \{A\}_K \\
\text{ThenB} &= B \rightarrow A : \{B\}_K \\
\text{SendData} &= A \rightarrow B : \{\text{Data}\}_K
\end{aligned}
$$

Figure 3.20: Agreement Extension Heuristic Example

Having constructed a possible extension, we still need the assisting lemmas for the final proof. We initially simply construct lemmas ensuring that the extension we have provided is a valid trace of the protocol. Theoretically these lemmas are guaranteed to be true, and thus proving them is not vital. However, to ensure the validity of the tool, and of our heuristic algorithm, we do prove them. This distinction is maintained in the PVS encoding through the use of *axioms* versus *lemmas*. We generally assume axioms will be true, without the need for proof. We instantiate these lemmas in the final proof for agreement when required to provide a valid extension. Apart from the extension lemmas, there is only one additional class of lemma produced for agreement: the requirement that the agreement token be at least partially concealed by encryption. While it is not a requirement in the strictest sense, it is highly unlikely that any protocol would ever be interested in the agreement of any non-encrypted token. If this were the case, these lemmas would simply have to be ignored.

3.6.2 Agreement Heuristic Limitations

While this technique works well for simple agreement tokens, it can be shown to be insufficient under certain more advanced protocols. We can demonstrate the flaw in the above heuristic algorithm using the test protocol defined in Figure 3.21.

$$\begin{aligned} \text{ExtensionFlaw} &= \text{SendEncrypted}; \text{SendKey} \\ \text{SendEncrypted} &= A \rightarrow B : \{\text{Data}\}_K \\ \text{SendKey} &= A \rightarrow B : K \end{aligned}$$

Figure 3.21: Extension Heuristic Flaw Example

In our example, we would heuristically guess that the message which revealed `Data` to host `B` would be `SendEncrypted`, since this is where the token is first seen. We would then generate the extension to this message only. However, this is not the message at which `B` would be able to agree with `A` on the token, as the encryption key is currently not known. Thus the actual extension we require for the proof would be all messages up to, and including, `SendKey`.

We investigated attempting to counter this by always delaying the extension until we have negotiated both the token and its key, but often the key for the agreement is either initial knowledge, or is negotiated in a fairly complex handshaking sequence. The effort involved in decoding these transactions is more suited to the theorem proving system, as it has all of the relevant encoding details available, whereas our translator does not contain very advanced protocol analysis abilities. Another possible method we might use is to simply delay our extensions until they have reached the end of the protocol. This, however, is also flawed, since we cannot guarantee that a given protocol will ever terminate; some are recursive. We desire minimal extensions, since the larger they are, the more effort we must expend in the theorem proving tool trying to prove them.

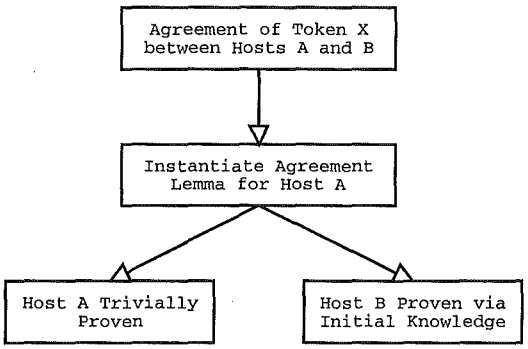
Thus we have used a heuristic technique, and have had satisfactory re-

sults. It should be noted that we can normally take our automatically generated extension and easily manipulate it in the theorem proving tool to generate a different extension. This ease of modification means that should we guess wrong in the translation, the semi-automated proof will fail. This will be visible to the operator, and will be easily identifiable as either a flaw in the extension generation, or possibly a flaw in the logic of the protocol relating to agreement.

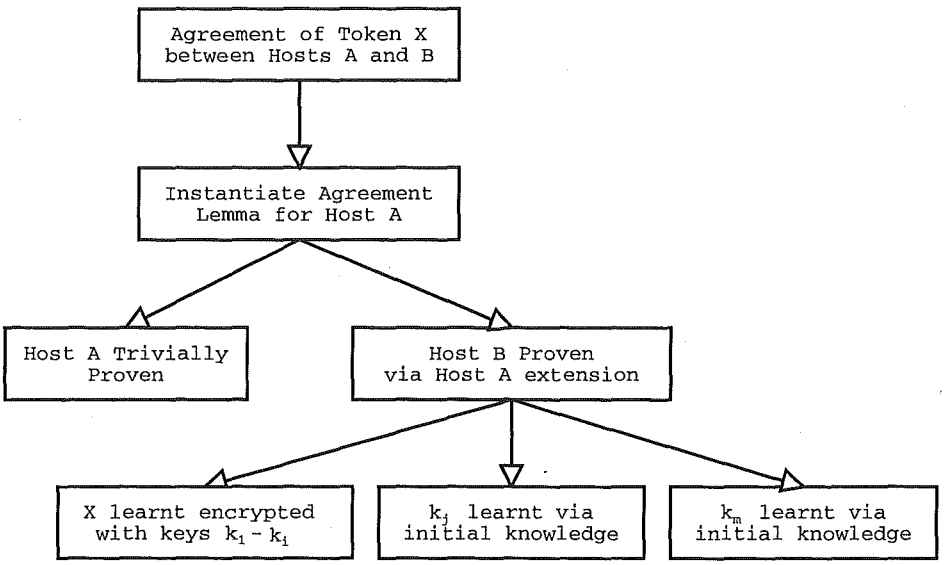
3.6.3 Global Agreement Theorem Structure

Our agreement theorem proof structure is not as rigidly defined as that for secrecy, as we now deal with two scenarios: either one of the agreement hosts knows the token using initial knowledge, or not. In the first scenario, the final theorem proof is quite simple, once the supporting lemmas have been completed. The structure of this is shown in Figure 3.22(a). Here, we can instantiate the agreement sub-lemma for the host who *does not* learn the token from initial knowledge, A in the case of the figure, which then trivially removes half of the work. The other half is then removed via the initial knowledge axioms for the other host, B in this case. This is the simpler of the two agreement scenarios.

In the second scenario, we have neither host knowing the token in their initial knowledge; both learn it from some third party. In this case, we can still use our agreement sub-lemmas, but additional work is required. The structure of this scenario is shown in Figure 3.22(b). Here, we must choose an extension in which *both* hosts can learn the token. This may be generated by the heuristic algorithm, but if it is not, it is not difficult to modify. Once this extension is generated, we can then use the sub-lemmas for agreement to show that for one of the two hosts, this extension allows the host to learn the given token. These sub-lemmas are simply used to break the proof down into more manageable chunks. They are not reused multiple times and so will not decrease the CPU requirements of a given proof. We can now use the appropriate sub-lemma to remove half of the agreement proof. Next we must prove that our extension allows the second host to learn the token of interest. For technical reasons we are unable to break this second proof



(a) Agreement Proof Scenario 1



(b) Agreement Proof Scenario 2

Figure 3.22: Agreement Proof Scenarios

off into a subsidiary lemma, and must prove it inline in the main theorem. Since there is little other work involved in this theorem, only a few protocol independent tree nodes, this does not interfere much, and is easily automated. This second proof may involve ensuring various keys are correctly learned for the host to learn the token, as we can see in the right-most leaf nodes of Figure 3.22(b).

There are two causes of difficulty in both the agreement sub-lemma proofs, and in the right leaf nodes of the final theorem proof. When the agreement token is found, it is most likely to be encrypted with some key or keys. This initially causes some difficulty. Each of these keys will have to be shown to be known to the host, either through initial knowledge, or via some other message. In some scenarios these keys may in turn be encrypted. This process can continue for some time before all are shown to be known. It is possible to break off the proofs for individual keys into sub-lemmas, but we have not done this automatically, as evaluating which keys are candidates for this involves considerable analysis of the protocol structure. We currently split these into separate sub-lemmas by hand if it is required.

3.7 Strategies

The auto-generation of supporting lemmas and the transition system provides the basis for our proofs, but does not minimise the effort of proving them. We require a scripting mechanism for this final step of automation. The theorem proving tool, PVS, allows us such a mechanism in the form of strategies.

PVS strategies are small sections of PVS prover commands, which also allow general LISP [Ste84] code. PVS contains all the infrastructure for implementing these directly into the proof system since many of the standard PVS commands are implemented as strategies using the core library. This leaves us with the task of generating our own appropriate strategies. Currently we have managed to write entirely generic strategies, not requiring any protocol specific information, thus allowing us to keep the same set of strategies for all proofs. However, it is likely that, as the system is extended, cases could arise in which generic strategies would be either impossible or too complex to be feasible. It would be a simple matter to add the generation

of these more complex strategies to the translation tool. Currently we have avoided auto-generation by supplying our strategies with optional arguments which define any protocol specific features in which we are interested.

3.7.1 Secrecy Strategies

The template we use for secrecy is quite simple. We continue to increase the depth of encryption that we are seeking, proving at each level that the token remains secret. At some point we will reach a depth of encryption which does not appear at all in the protocol, and thus the remainder of the proof is trivially true. This structure has already been discussed in Figure 3.17. We have already broken this down into a series of lemmas, as discussed earlier in Section 3.5, and thus we can use these previously proven lemmas to solve each of the branches. The initial, zero-level encryption depth branch is easily solved using the auto-generated `reveals` lemma for the token of interest. For example, using the protocol shown in Listing 3.1, we would have the following zero-level encryption depth lemma auto-generated for the secrecy of `Y` between hosts `A` and `B`:

"Y safe at zero-level" Lemma :

$$\forall\, T : \text{Trace} : \neg\text{reveals}(T, Y)$$

```
Protocol "SampleProtocol"

Variables
        A, B : Host
        KA, KB, KC : Key
        X, Y : Natural

Initial
        Knows(A) = KA, KA', KB, KC
        Knows(B) = KA, KB, KB', KC

Spec
```

```
        Secret(Y, [A, B])

SampleProtocol = Msg1 ; Msg2 ; Msg3
Msg1 = A -> B : X, {Y}KA'
Msg2 = B -> A : {Y}KB'
Msg3 = A -> B : {Y, X}KC
```

Listing 3.1: Sample Protocol

This is proven automatically via two other lemmas. The first states that all messages in a transaction must be valid protocol messages, since in our model the Spy is passive, and the second states that no valid message reveals Y. This second lemma is proven in terms of the secrecy lemmas for each individual message. We can prove the individual message theorems using a simple strategy. This strategy iterates through the body of the message, and ensures that none of the plain text tokens in the message are the token of interest. Since this does not involve encryption, this can be done very simply. The PVS code for this strategy is shown in Listing 3.2.

```
(defstep message_reveals (msg)
(then
        (skosimp)
        (rewrite msg)
        (replace -1)
        (repeat* (rewrite "reveals" -2))
        (assert)
        )
        "message_reveals" "message_reveals")
```

Listing 3.2: Message Reveals Strategy

The message reveals strategy is quite simple; it instantiates the message, and rewrites it to bring the message details into a manageable place. It then repeatedly removes the head of the list of tokens, thus creating a conjunction of boolean values corresponding to whether a certain token in the list was

the token of interest. Since we are hoping that this is never the case, this will simply be a list of `false` items, and thus the final assert will reduce this conjunction to the single term `false`, and the proof is then complete.

The use of the constants, -1 and -2 denote statement numbers to be manipulated, as discussed in Section 2.2.1. They can be removed through the use of the PVS command `gather-seq`, which can perform a form of pattern matching to discover which particular statements are desired. However, in the case of most of our strategies the scenarios are quite limited, and thus do not vary enough for these constant values to cause problems.

This is actually the most complex strategy for the zero-level encryption branch, since we now no longer have to deal with the specific details of the contents of a protocol's messages. Once we have proven that no individual message reveals the token we can then prove that, since all valid traces of the system are made up of these messages, no valid trace can reveal the token. We will not go into the details of this strategy. It is simply a matter of instantiating all the individual message lemmas.

The zero-level encryption lemma is now proven, and can be used in the secrecy theorem strategy to solve the left-most branch of the proof. We then have n lemmas auto-generated for the depth of the protocol. Thus, in our example from Listing 3.1, we only have a single depth of encryption, that is, there are no messages which contain nested encryption. We would therefore generate only one lemma relating to encryption depth, as follows:

"Y safe at depth 1" Lemma :

$$\begin{aligned}
&\forall\, T : Trace, K : Key, F : Field : \\
&\quad \text{reveals}(T, F_K) \Rightarrow \\
&\qquad \neg\text{reveals}(F, Y) \vee \\
&\qquad \neg\text{knows}(S, T, K)
\end{aligned}$$

For the i-level lemmas in general, the strategy for their proofs is as follows. Initially, we break down the trace into the individual message components, that is, if the protocol reveals something at 1-level, then there must be at least one message which reveals it at this depth. Once we have this, we then

have n branches to the proof, one for each message in the protocol. Given a trace T, and a token of interest F, each of these branches would be of the following form:

$$\begin{aligned}
&\text{reveals}(Message_1, \{Y_1\}_{K_1}) \wedge \\
&\text{reveals}(Y_1, \{Y_2\}_{K_2}) \\
&\quad \vdots \\
&\text{reveals}(Y_{i-1}, \{Y_i\}_{K_i} \Rightarrow \\
&\quad \neg\text{reveals}(Y_i, F) \vee \\
&\quad \neg\text{knows}(\text{Spy}, T, K_1) \vee \\
&\quad\quad \vdots \\
&\quad \neg\text{knows}(\text{Spy}, T, K_i)
\end{aligned}$$

These portions can also be broken off into sub-lemmas for each message, which state that, for each message, either the i-level encryption does not appear in the body of the message, or the keys to the encryption are never known. We have currently only partially automated the proofs for these lemmas. Our strategies will currently automatically prove all cases where the above `reveals` example does not apply, that is, if a message does not reveal the given token in i levels of encryption, then it is proven automatically. If this fails, then the operator is left to complete the proof. This partial automation does, however, quickly highlight the core of the problem, so that the operator does not have to deal with trivial cases. Using all of these sub-lemmas, it is now simple to instantiate them appropriately on each of the n branches. Thus, for i-level encryption we currently only offer a partially automated analysis. The strategy we use for i-level is very similar to that for zero-level, and is shown in Listing 3.3.

```
(defstep i_level_reveals (msg)
(then
        (skosimp)
        (rewrite msg)
```

```
        (replace -1)
        (repeat* (rewrite "reveals" -2))
        (then
                (split)
10              (repeat* (flatten))
        )
        )
        "i_level_reveals" "i_level_reveals")
```

Listing 3.3: i-level Secrecy Strategy

In this strategy we simply break down the message into a list of candidate tokens. These candidates are simply encrypted tokens. We then split this into a set of leaf nodes which need to be proven. If there are no candidates (as in the trivial case), then the proof would be complete. For example, if we were analysing a 3-level encryption depth, we would break the message down, discovering all of the 3-level encryption tokens, which would then be presented to the operator. We can see that this removes the tedium from the operator, leaving only likely candidates to be dealt with.

The main difference between this strategy and the zero-level one is in the `(then ...)` block. This block simply iterates through every leaf node of the system, flattening all boolean statements. This is used to convert statements of the form `TRUE ^ FALSE ^ TRUE` into `FALSE`, and trivially eliminates all those which fail. This removes most of the tedium from the operator, as all of the trivial cases are automatically proven.

Finally, as we have previously discussed, we would also generate the terminating lemma, ensuring that our recursive definition of `knows` will have a finite depth. This is defined as:

"SampleProtocol has maximal depth of 1" Lemma :

$$\begin{aligned}\forall\, T : \text{Trace}, K_1, K_2 : \text{Key}, Y_1, Y_2, Y_3 : \text{Field} :\\ \text{reveals}(Y_3, \{Y_2\}_{K_2}) \wedge\\ \text{reveals}(Y_2, \{Y_1\}_{K_1}) \Rightarrow\\ \neg Y_3 \in \text{parts}(T)\end{aligned}$$

In summary, we can thus automatically prove the zero-level encryption lemma, and the maximal encryption depth lemma, and have partially automated the i-level encryption depth lemmas. Once these are proven we can trivially prove the final secrecy theorems.

3.7.2 Agreement Strategies

Agreement does not break down into such a simple structure as secrecy, because it is a liveness, rather than safety, property. As a liveness property, agreement does not necessarily hold true over the entire run over a protocol, only in specific states. Thus the first task we must attempt is to find these states. Thus is currently done heuristically, as discussed in Section 3.6. In our strategies, a completely automated technique for dealing with agreement has yet to be developed. The fundamental problem we currently face is the size of the proof trees. Agreement involves a large amount of rewriting, because we have a specific trace in mind with which we are dealing, and thus we have to maintain a large amount of code for dealing with this trace. Since it is a specific trace instance, rather than an abstract instance, we have a large number of tokens in our tree in which we are not particularly interested, but must retain to ensure the integrity of the trace. These tokens are generally messages which do not contribute at all to the agreement property. Normally in a trace there are one, possibly two, messages of interest for a specific agreement property. However, the trace to reach these messages may consist of as much as the entire protocol, which, in the case of a large protocol such as SET, would be too much for the computer to work with comfortably.

The overwhelming amount of useless data this causes is currently handled simply by manually hiding all data which is of no interest. Unfortunately, we currently have no automated technique for guessing what is useful, and so this requires human intervention.

However, although the overall structure of the proof remains unscripted, we do have a model for it, and have constructed strategies for proving both its subsidiary lemmas, and portions of the final proof. Most of the lemmas used in agreement are shared with secrecy, as most of the time we are simply informing the system that a given message does not contain the token in

which we are interested. We are thus left with a set of candidate messages. One of these should be the message which reveals the token to the target host. Therefore two classes of messages remain: messages which reveal the token as desired, and those which contain it, but do not reveal it to the target host. The overall focus of our strategies has generally been to attempt to automatically prove sections which consist of either mundane details and thus are of little interest to the user, or are very frequently used and thus constitute a large portion of the overall workload. In this case, using a breadth first search, we have targeted the messages that do not reveal the given token, as they are of little interest to the user. However, since our trace is not guaranteed to be finite in length, we cannot use the standard brute force technique. Therefore a guided technique is used, in which the operator must choose the appropriate path.

In summary, high level strategies for dealing with agreement are yet to be developed, but we do have a wide variety of small, specific strategies for simplifying the operator's interaction.

Chapter IV

Case Studies

4.1 General Comments

The design of our translation system has been tested using two protocols, NetBill and SET, as case studies to determine appropriate properties of interest. Two protocols have been used to ensure that the system was not heavily biased towards one style. Preliminary analysis of two other protocols, Millicent [Man95] and IKE [HC98], was also begun. However, the analysis of these is incomplete compared to the other two, and therefore is not included in this thesis. Both of the chosen protocols are e-commerce protocols, and are designed for the transferring of either money or goods in a secure way. We do not believe that this has caused any form of bias in these results towards e-commerce protocols, since, due to the abstraction of the specific data involved, they are very similar to more standard security protocols, such as IKE.

4.2 Secure Electronic Transaction – SET

The SET [VIS97] protocol is a highly specialised, secure, e-commerce system. It was jointly designed by VISA and Mastercard, as well as a number of other large computer industry players, such as Microsoft and Intel. It is one of the largest secure transfer protocols with respect to the number of message variations. The full definition of SET is just under 1000 pages, with 260 pages for the protocol description, 630 pages for the programming description, and 80 pages for a business management overview. Only a small subset of the full protocol has been analysed for our case study.

SET is often compared to the TLS [DA99] protocol when used for e-

commerce style business applications, although it differs in its fundamental design concepts. SET uses highly selective encryption [TY98], whereas TLS is a data independent protocol, and, as such, encrypts all data, providing a secure stream. SET uses only partial encryption to provide for higher throughput at a lower per transaction cost.

SET is a highly application specific protocol, with separate message handshakes for each of its numerous tasks. These tasks are very fine grained; for example it contains a set of messages relating specifically to booking rooms in hotels. This fine-grained nature is what makes SET so large, and also makes it difficult to model as an entire system.

The notation used in the SET description differs from the standard protocol description notation, and a high degree of familiarity with the protocol and its conventions is required to convert the standard description into the CAPSL-like syntax required by our translator. This conversion took approximately sixteen hours for the subset we describe in Section 4.2.1. However this includes the initial difficulties with the SET description language, and would thus be lower for subsequent encodings. The SET description is also broken down into very small logical chunks, which are reused many times. This fragmentation means that we would also be able to reuse sections of our encoding in subsequent descriptions, which would decrease the time and effort involved in converting from their input language to ours.

4.2.1 SET Protocol Description Encoding

The full description of the SET protocol subset we have used is shown in Listing 4.1. Initially all public keys, `KC` and `KM`, are known to all hosts, private keys `KM'` and `KC'` are known to their respective hosts only, and `CardID`, the secret credit card number is known only to the customer, `C`. These features are shown in the `Initial` section of our description. We have three security properties of interest in SET, all of which relate to secrecy. The first two are both simple key secrecy, that private keys should remain private. The final secrecy property relates to the semantics of the protocol itself, that `CardID` should remain a secret between the merchant and the customer. These are all shown in the `Spec` section of the description.

```
Protocol "SET"


Variables
   KM, KC : Key
   ID, CardID, Amount, ReqID, ChallengeC,
                  ChallengeM, BankNo: Natural
   SigC, SigM: Signature
   DD, SHA : Hash
   EXNonce : Nonce
   C, M : Host

Initial
   Knows(M) = KM, KC, KM'
   Knows(C) = KM, KC, KC', CardID
   Knows(S) = KM, KC

Spy
   Observation, Synthesis

Spec
   Secret (KC', C)
   Secret (KM', M)
   Secret (CardID, [M, C])

# Payment Messages (See Page 71 of SET manual)

SET = PInitReq ; PInitRes ; PReq

PInitReq = C -> M : ReqID, ChallengeC, BankNo
PInitRes = M -> C : ChallengeM, {ChallengeM}SigM
PReq = PReqDualSigned + PReqUnsigned

# Dual Signed
PReqDualSigned = PIDualSigned ; OIDualSigned

# PIHead: Page 37
# PIHead = ID, Amount, CardID
# OIData = ID, SHA(Amount)
# PI-OILink = PIHead, DD (OIData) == L (PIHead, OIData)

# PIDualSigned Page 35
PIDualSigned = C -> M : SigC(C), {ID, Amount, CardID,
```

```
                                        DD (ID, SHA (Amount)),
45                                      CardID, EXNonce}KC'
  # OIDualSigned: Page 77
  OIDualSigned = C -> M : ID, SHA (Amount),
                          DD (ID, Amount, CardID)

50 # Unsigned
  PReqUnsigned = PIUnsigned ; OIUnsigned
  PIUnsigned = C -> M : {ID, Amount, CardID,
                         DD (ID, SHA (Amount)),
                         CardID, EXNonce}KC'
55 OIUnsigned = C -> M : ID, SHA (Amount)
```

Listing 4.1: SET Protocol Description

4.2.2 Protocol Details

The intuitive meanings of each of the protocol messages are defined as follows:

SET This is a process element describing the structure of the protocol. In this case it shows how the three sub-components fit together in a simple sequence.

PInitReq Here we describe the initial request from the customer. This involves request ID, which identifies the goods in which we are interested, a challenge, which is simply a list of supported protocols for items such as encryption and hashing, and finally the bank number, which is used as a customer identification number in the SET protocol.

PInitRes Here the merchant responds to a customer's request. The merchant simply responds with the protocols (for hashing and encryption as discussed above) which the pair will use for the remainder of the communication. As we have assumed all hashing and encryption to be perfect, we are not interested in the actual details of this as we consider all of these algorithms to be equal.

PReq Here we describe the final two message exchanges, which can come in two forms: signed and unsigned. This specifically relates to whether

the two hosts have a third signature authority available. This is specific to the circumstances of the transaction, and either may be used. This message also demonstrates the use of the non-deterministic choice feature of our input language.

PReqDualSigned This is a simple structure node. Since the `PReq` portion of the protocol is a two message communication, we simply join these two messages to make the `PReqDualSigned` process.

PIDualSigned The customer informs the merchant of the particulars of the order, such as the amount of the purchase and the credit card number to use. Since this is the signed option of the protocol, the signature of the customer is attached.

OIDualSigned This message is sent to validate the `PIDualSigned`. It contains a hash of some of the data previously sent, which the merchant can compute locally and then compare to the sent value to ensure the messages were sent safely.

PReqUnsigned Similarly to `PReqDualSigned`, this process is a simple structure process denoting the two messages of the unsigned half of `PReq`.

PIUnsigned This message is identical to `PIDualSigned`, except the customer's signature is omitted.

OIUnsigned This message is identical to `OIDualSigned`, except the digital digest hash is omitted.

4.2.3 Secrecy Lemma and Proof Examples

Some examples of some of the lemmas generated by our translator for the SET protocol are discussed below. The two secrecy properties relating to the keys `KC'` and `KM'` are of little interest since we have not modelled the key exchange section of SET, but we will examine the encoding of the secrecy of `CardID`, the credit card number, in depth. The lemmas we will discuss are

shown in Figure 4.1. Initially we require the lemmas relating the secrecy of `CardID` to each specific message, as discussed in Section 3.5.1. An example of one of these lemmas is shown in Figure 4.1(a). From the eight message dependent lemmas, we then generate a generalised lemma about all valid messages. This is, in turn, used to prove a lemma relating to all traces, shown in Figure 4.1(b). This is the zero-level encryption lemma we require for the first step of our proof. SET only contains 1-level encryption, and so we only require a single i-level encryption lemma, shown in Figure 4.1(c). The encryption bounding lemma is not token specific, and so there would be only one instance of this lemma type for the entire protocol, rather than the three we require for other lemmas. The SET instance of this encryption bounding lemma is shown in Figure 4.1(d). Thus, these final three lemmas can now be used as the leaf nodes in our SET secrecy proof, the structure of which was discussed in Section 3.5.2.

Key Secrecy in `knows`

When dealing with the i-levels of encryption, as shown in Figure 4.1(c), there are two possible cases for any given message type: either the given token never occurs at the given depth, or it does appear, but the key with which it is encrypted is never known. The first of these two cases is proven simply by breaking down the message into its individual components, none of which are the token of interest, and thus the lemma is trivially true. The second case is that which takes the majority of the proving effort. In this case it is the statement, $\neg knows(S, T, K)$ which we must prove to be true, or rather $knows(S, T, K)$ must be proven false. This is again proven recursively, just as we are attempting to prove the overall secrecy theorem for `CardID`. The zero-level of encryption is shown through our key secrecy lemmas, discussed in Section 3.5.1, and the i-level proofs are conducted in exactly the same manner as for any other secret token, i.e.: `CardID`. Thus in the case of candidate messages, that is, messages which contain the given token at the correct encryption depth, we are required to perform another almost identical proof, this time on the key of interest. This is seen in the `PIDualSigned` message for SET, where CardID appears encrypted with the

"CardID safe in OIUnsigned" Lemma :

$$\neg\text{reveals}(\text{OIUnsigned}, \text{CardID})$$

(a) CardID secrecy in OIUnsigned message

"CardID safe at zero-level" Lemma :

$$\forall\, T : \text{valid}(T) \Rightarrow \neg\text{reveals}(T, \text{CardID})$$

(b) CardID secrecy at zero-level encryption

"CardID safe at 1-level" Lemma :

$$\forall\, T, K, Y : \text{valid}(T) \wedge \text{reveals}(T, \{Y\}_K) \Rightarrow \neg\text{reveals}(Y, \text{CardID}) \vee \neg\text{knows}(S, T, K)$$

(c) CardID safe at 1-level encryption

"SET limited to depth 2" Lemma :

$$\forall\, T, K_1, K_2, Y_1, Y_2 : \text{valid}(T) \wedge \text{reveals}(Y_2, Y_{1K_1}) \Rightarrow Y_{2K_2} \notin \text{parts}(T)$$

(d) SET contains maximal encryption depth

Figure 4.1: Some SET CardID Secrecy Lemmas

KC' key, which must then be proven for secrecy. Apart from the zero-depth portion, these key secrecy proofs are not currently extracted into full sub-lemmas. In general there will only be a few keys which are of interest to a given token, and only a few messages in which they apply, so the benefits of proving them elsewhere are unlikely to be large. If we wished this it would be quite simple, since we desire almost all non-public keys to be secure. We could simply generate secrecy lemmas for these automatically, but most of the cases would be of no interest to the end user. This behaviour can also be emulated by adding secrecy lines for all of the keys to the input file, thus explicitly stating their secret nature. This is the preferred mechanism for keys, since it allows the operator to generate only the lemmas for keys which are shown to be of interest.

Since the output PVS code from the translator does not need to be edited by the operator, it is a simple matter to begin a proof, and then if a key is discovered which must be proven secret, a single line could be added to the protocol definition, and the output regenerated. PVS maintains its proofs in a separate context, so the proof structure for all previously proven lemmas and theorems would still be available.

Secrecy Breakdown

To demonstrate how the earlier discussion of proof structure relates to the actual user interaction with the PVS theorem proving environment, we have included the proof tree which PVS generates after our proof of the KC' secrecy. This can be seen in Figure 4.2. If we recall the general proof model, shown in Figure 3.17(a), we can see that our actual proof is very similar in structure. We have three basic branches. The left-most, with the `KC_valid_trace_safe` node, denotes the zero encryption/reveals level. The central branch denotes the single level of encryption that SET contains. The final branch denotes the maximal encryption branch, in which we state that no encryption greater than one will help, since our SET subset is bounded to one level encryption. We have hidden some of the PVS details behind small strategies, which are denoted by the nodes labelled `knows_reduce`, and `enc_knows_reveals`. These are not complex, only four – five proof tree

nodes, but we have removed them as they detract from the visibility of the overall structure. As can be seen, the final secrecy proof is very short once the supporting lemmas and strategies are in place. This view also makes it clear which lemmas we have relied on for the proof, since they are explicitly mentioned with the `lemma` command.

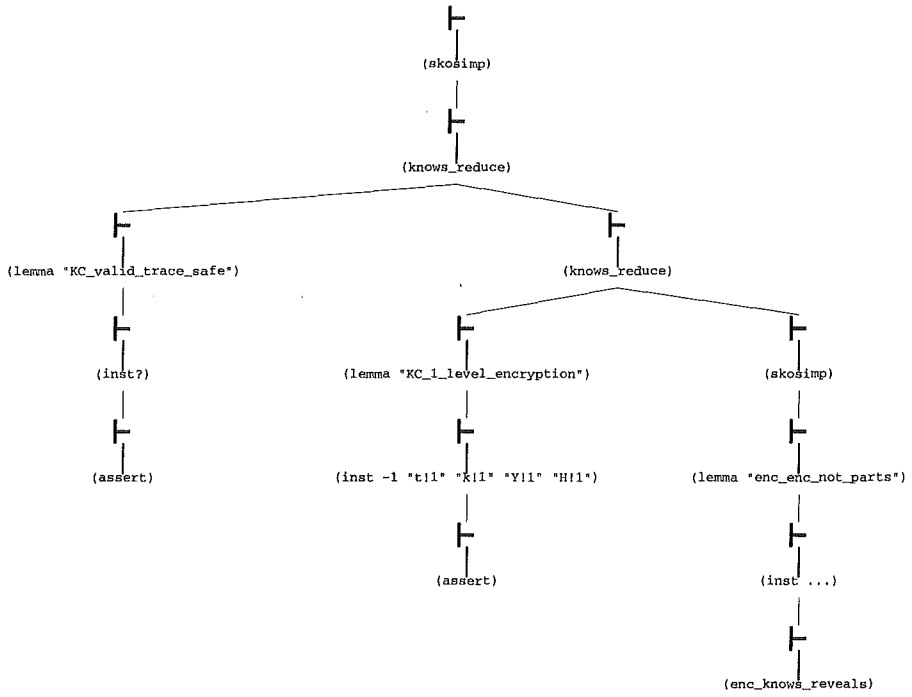


Figure 4.2: SET PVS Proof Tree

Since this proof no longer has to know the exact details of the messages involved, we have developed a simple strategy for the final secrecy proof. We simply take the token of interest, `KC`, and the depth of the protocol, 1 in the case of SET, instantiate the 0-depth lemma, iterate through the depth of the protocol instantiating each of the various depth lemmas, and then finally instantiate the bounded encryption depth lemma. This strategy is quite long, but does not contain any interesting details.

4.3 NetBill

NetBill [CTS95] is a micro-payment system, designed to allow for payments of fractions of a cent to be cost effective. In traditional secure payment protocols, the cost of performing the computation and transaction makes low valued transactions uneconomical. NetBill attempts to address this by allowing for a highly simple, yet secure protocol, designed for one-off transactions of a small nature. The NetBill description took less than a day to convert into our system, the core protocol definition being only sixteen pages. We have not encoded some of its optional extras, such as zero cost transactions and the status query exchange, but we have modelled all of the base protocol.

Originally developed by Carnegie Mellon University, NetBill was later sold to Cybercash, and is now available commercially from `http://www.cybercash.com/`

4.3.1 NetBill Protocol Description Encoding

```
Protocol "NetBill"

Variables
   C, M, N : Host
   K_CN, K_CM, K_MN, K_K : Key
   IdentityN, IdentityC, IdentityM, TID, EPOID : Nonce
   MAcct, PRD, Bal, CAcct, Price, Goods : Natural
   CC : Hash
   SigN, SigM, SigC : Signature

Initial
   Knows(C) = K_CM, K_CM', K_CN, K_CN'
   Knows(M) = K_CM, K_CM', K_MN, K_MN', K_K, Goods, EPOID
   Knows(N) = K_MN, K_MN', K_CN, K_CN'

Spy
   Observation, Synthesis

Spec
   Agreement (C, N, EPOID)
   Agreement (C, M, Goods)
```

```
    Secret (CAcct, [C, N])
    Secret (MAcct, [M, N])

25
  NetBill = PriceRequest ; GoodsDelivery ; PaymentPhase

  PriceRequest = Msg_1 ; Msg_2

30 Msg_1 = C -> M : {IdentityC}SigC, {PRD}K_CM

  Msg_2 = M -> C : {PRD, Price, TID}K_CM

  GoodsDelivery = Msg_3 ; Msg_4

35
  Msg_3 = C -> M : {IdentityC}SigC, {TID}K_CM

  Msg_4 = M -> C : {Goods}K_K, {{{Goods}K_K}CC, EPOID}K_CM

40 PaymentPhase = Msg_5 ; Msg_6 ; Msg_7 ; Msg_8

  Msg_5 = C -> M : {IdentityC}SigC,
                   {{IdentityC, PRD, Price, EPOID}SigC}K_CM

45 Msg_6 = M -> N : {IdentityM}SigM,
                   {{{IdentityC, PRD, Price, EPOID}SigC,
                                        MAcct, K_K}SigM}K_MN

  Msg_7 = N -> M : {{IdentityN, PRD, K_K, EPOID}SigN,
50                  {EPOID, CAcct, Bal}K_CN}K_MN

  Msg_8 = M -> C : {{IdentityM, PRD, K_K, EPOID}SigN,
                   {EPOID, CAcct, Bal}K_CN}K_CM
```

Listing 4.2: NetBill Protocol Description

4.3.2 Protocol Details

NetBill This is a simple structure message indicating how the three basic sections of NetBill are composed. Since this is a simple linear protocol, the three sections appear consecutively.

PriceRequest For ease of understanding, and to comply with the structure

of the documentation for NetBill, we divided the protocol into three sub-protocols. PriceRequest is the first of these.

Msg_1 Here the customer informs the merchant of their identity, as well as the product item to be purchased.

Msg_2 The merchant responds to the customer with the price, and a unique transaction ID.

GoodsDelivery This is the second sub-protocol. It indicates the encrypted goods exchange.

Msg_3 The customer here confirms the transaction ID with the merchant.

Msg_4 Here the encrypted goods are sent to the customer, as well as a unique episode ID, which is used to identify the session with the NetBill server.

PaymentPhase This is the third sub-protocol. The NetBill server is introduced, and is used by both the merchant and the customer to ensure that the transaction is authentic.

Msg_5 The customer sends the appropriate tokens for the purchase order to the merchant. The tokens vital to the transaction are also signed by the customer. This is the last opportunity for the customer to abort the transaction.

Msg_6 The merchant adds their own details to the customer's signed purchase order, and forwards it on to the NetBill server. This is the key message in the protocol, as, at any time before this, the merchant could have aborted the transaction for any reason. It should be noted that this message contains sections signed by both customer and merchant, so is quite secure from forgery.

Msg_7 The NetBill server now responds with useful information to the merchant, including information on what money has been transferred.

Some of this information is encrypted with a key which the merchant cannot reverse, so details such as the customer account and balance are not revealed.

Msg_8 Here the merchant forwards the NetBill server's customer information back to the customer, as well as the decryption key, also signed by the NetBill server.

An interesting point about the NetBill Protocol is its use of encryption and signatures to forward information from the NetBill server through the merchant to the customer. This forwarding is done by using keys of which the merchant has no knowledge, to prevent the merchant from being able to inspect their contents. Message 8 contains an example of this. Thus, although the customer never directly interacts with the NetBill server, they can still verify that the results have not been tampered with, and can be assured that the communication is indeed secure.

4.3.3 Agreement Lemma and Proof Examples

We now analyse how the agreement properties were dealt with in NetBill. In particular, we are interested in the extension generation.

As can be seen from the NetBill protocol description in Listing 4.2, we require agreement between the customer and the NetBill server (`C` and `N`) on the `EPOID` token. `EPOID` is a unique Episode Purchase Order ID. The first step in the protocol dependent section of agreement auto-generation is to develop a suitable extension in which we believe agreement occurs. From the protocol description we can see that the NetBill server, `N`, first learns of the `EPOID` in `Msg_6`, when it is received from the merchant. Our heuristic thus guesses at `Msg_6` as the end point for the NetBill server extension. From the customer perspective, the first message in which `EPOID` is discussed is `Msg_4`, and this would thus be the end point for that extension. Due to the simple linear nature of the NetBill protocol, the backtracking is trivial. The final extension guess for the NetBill server is thus:

$$\texttt{Msg\_1} \rightarrow \texttt{Msg\_2} \rightarrow \texttt{Msg\_3} \rightarrow \texttt{Msg\_4} \rightarrow \texttt{Msg\_5} \rightarrow \texttt{Msg\_6}$$

and for the customer:

$$\texttt{Msg\_1} \rightarrow \texttt{Msg\_2} \rightarrow \texttt{Msg\_3} \rightarrow \texttt{Msg\_4}$$

The lemmas to generate these extensions are shown in Figures 4.3(a) and 4.3(b). These lemmas simply state that a given sequence of messages is a valid extension. They do not state anything about this extension, merely that it remains a valid run of the protocol. Thus, even if these are not the extensions of interest, due to the discussed flaw in our heuristic, we are easily able to modify them without affecting the remainder of the system.

"Extension for EPOID and N" Lemma :

$$\forall\, T : \exists\, E :$$
$$E = [T, \mathrm{Msg}_1, \mathrm{Msg}_2, \mathrm{Msg}_3, \mathrm{Msg}_4, \mathrm{Msg}_5, \mathrm{Msg}_6] \wedge$$
$$\mathrm{validExtension}(T, E)$$

(a) NetBill Server Extension

"Extension for EPOID and C" Lemma :

$$\forall\, T : \exists\, E :$$
$$E = [T, \mathrm{Msg}_1, \mathrm{Msg}_2, \mathrm{Msg}_3, \mathrm{Msg}_4] \wedge$$
$$\mathrm{validExtension}(T, E)$$

(b) Customer Extension

Figure 4.3: NetBill Agreement Extension Generation Lemmas

Using these two lemmas, we can now instantiate these two specific runs into our general theorem. The general agreement theorem for this property is shown in Figure 4.4.

We also break down the general lemma into two subsidiary lemmas, one for each host. Thus we prove that each individual host can learn the appropriate token, `EPOID` in this case, and then use these proofs in the final

"C and N agree on EPOID" Theorem :

$$\forall\, T : \exists\, E :$$
$$\text{validExtension}(T, E) \wedge$$
$$\text{knows}(C, E, \text{EPOID}) \iff$$
$$\text{knows}(N, E, \text{EPOID})$$

Figure 4.4: NetBill EPOID Agreement Theorem

agreement theorem. The NetBill instance for the NetBill server, `N` is shown in Figure 4.5. The lemma for the customer, `C`, is omitted, although it would be almost identical to that in Figure 4.5 except the final `N` would be a `C`. In this example, T represents the current trace of the protocol, while E represents the extension that we are attempting to prove satisfies our agreement criteria.

"N learns EPOID" Lemma :

$$\forall\, T : \exists\, E :$$
$$\text{knows}(N, e, \text{EPOID}) \wedge$$
$$\text{validExtension}(T, E)$$

Figure 4.5: NetBill EPOID Agreement Host Lemmas

4.3.4 Goods Agreement: Proof Difficulties

As we discussed in Section 3.6.2, there are protocols in which our agreement extension generation heuristic does not generate a suitable possibility. This is the case in the NetBill protocol for the `Goods` token. Here, the `Goods` token is sent to the customer in `Msg_4`; although the key is not divulged until `Msg_8`. This flawed heuristic causes our auto-generated extension to be useless. Thus, for our proof of Goods agreement, we were required to manually build the extension in which we were interested. In the case of

NetBill, a purely linear protocol, this was not difficult, as we simply added all messages from `Msg_4` onwards to the end of the extension. However, in a larger non-deterministic protocol this could conceivably cause problems for the operator.

4.3.5 NetBill Secrecy

As shown in the NetBill protocol description, Listing 4.2, we have two tokens which we require to remain secret, `CAcct`, and `MAcct`, the customer account number, and the merchant account number. We will not describe these proofs in depth, as they are very similar to the SET `CardID` proof. NetBill, however, has one important difference to SET with respect to secrecy proofs. It contains two levels of encryption, in messages `Msg_4`, `Msg_7` and `Msg_8`. This means that our final proof will have four leaf nodes, rather than three: zero-level encryption, 1-level encryption, 2-level encryption, and encryption bounding. The 2-level encryption, and encryption bounding lemmas for `CAcct` are shown in Figures 4.6(a), and 4.6(b) respectively.

4.4 Effort Analysis

Two hundred theorems were required by [BMPT00] in their analysis of Cardholder Registration in SET. However, it is not directly comparable to our system, since they use both a different encoding and theorem proving tool. The theorem proving tool used, Isabelle, requires a much deeper embedding than PVS, and would, as such, require a larger number of supporting theorems.

Using the NetBill case study, specifically the `CAcct` secrecy, we can conduct some analysis on the relative effort before and after the use of our automatic lemma and strategy generation. Performing one of the secrecy proofs without any of our lemmas or strategies took approximately twelve hours of interaction with the system, although according to the system logs, only six hours of that was on computation, the remainder was idle time during operator interaction with the system. This number is still not too dependable, since we believe that the system began swapping memory pages to disk due to the huge size of the proof trees. This would have increased the time it

"CAcct safe at 2 level" Lemma :

$$\begin{aligned}
&\forall\, T, K_1, K_2, Y_1, Y_2 : \\
&\text{valid}(T) \wedge \\
&\text{reveals}(T, \{Y_2\}_{K_2}) \wedge \\
&\text{reveals}(Y_2, \{Y_1\}_{K_1} \Rightarrow \\
&\qquad \neg\text{reveals}(Y_1, \text{CAcct}) \vee \\
&\qquad \neg\text{knows}(S, T, K_1) \vee \\
&\qquad \neg\text{knows}(S, T, K_1)
\end{aligned}$$

(a) CAcct safe at 2-level encryption

"NetBill limited to depth 3" Lemma :

$$\begin{aligned}
&\forall\, T, K_1, K_2, K_3, Y_1, Y_2, Y_3 : \\
&\text{valid}(T) \wedge \\
&\text{reveals}(Y_3, Y_{2K_2}) \Rightarrow \\
&\text{reveals}(Y_2, Y_{1K_1}) \Rightarrow \\
&\qquad Y_{3K_3} \notin \text{parts}(T)
\end{aligned}$$

(b) NetBill contains maximal encryption depth

Figure 4.6: Some NetBill CAcct Secrecy Lemmas

took to perform the computation, and since our simplified structure is small enough to avoid swapping, this slow down will not have been apparent. Thus minimising memory requirements is another benefit of using our system.

Once we have our auto-generated lemmas, the final proof is performed in under a minute. However, since we have now split the proof into multiple lemmas, this statistic is obviously skewed as we are now working on a reduced problem. Thus if we take into account the proof time for all the sub-lemmas, we would have the new estimate of effort. We will ignore interaction time in these numbers, as this is difficult to measure in an operator independent way.

The simple key secrecy and token secrecy, as discussed in Section 3.5.1, are now proven automatically via strategies, and take approximately half a second each. For a single secrecy property analysis, we have a key secrecy lemma and a token secrecy for each message. Thus for the 8-message NetBill protocol, we would have sixteen lemmas generated in total for these two categories. All of these lemmas would thus take a total of four seconds of CPU time to prove. We currently prove these lemmas interactively, so each lemma must be selected, and then the strategy applied, thus taking additional interaction time. However a PVS batch mode is available in which strategies and partial proofs can be attached to lemmas. This batch mode allows us to associate the lemmas and strategies appropriately, thus removing all user interaction from these proofs. We have not investigated this too deeply however. Since our proof is not entirely automated, there will always be a need to enter the interactive mode to perform the final proof, and thus the benefits of proving half of the lemmas in batch mode, and the remainder interactively are not large. Were we able to fully automated secrecy, it would be worth investigating the use of the PVS batch mode for this.

For 1-level encryption we took one second to prove that for each individual message, no key was revealed in one level, and one second to prove the general lemma, although for NetBill this may be slightly biased, since CAcct never appears within one-level of encryption (only two-levels). For two-level encryption, which is of interest for NetBill, as `CAcct` appears doubly encrypted in both messages `Msg_7` and `Msg_8`, we took one second for the six lemmas which did not contain two-levels of encryption, and 45 seconds for the two that did. This is an average of twelve seconds each. Once these lemmas were completed, the general two-level encryption lemma proof took only two seconds. Finally, for maximal depth, the eight individual lemma proofs took two seconds, with the final generalised proof also taking two seconds.

The total time taken is thus 129 seconds, slightly over two minutes. While this time relates strictly to the computational effort, and not user interaction, it is still a significant improvement over the manual encoding. As can be seen, the majority of the work is currently at the i-level encryption stage. This is

where we hope to develop additional strategies. It should also be noted that the message encryption depth lemmas and the key secrecy lemmas are reused between all of the secrecy protocols. These account for eighteen seconds of the total time.

Although we have used the computation time, rather than interaction time, we normally see a ratio of approximately 10:1 in lemmas whose strategies are incomplete, such as the 2-level encryption, and a ratio of only 1.2:1 in those lemmas that have full strategies. We would thus anticipate taking somewhere in the region of fifteen minutes of real operator time to conduct these proofs. This figure is still significantly smaller than that seen in the original non-automated system.

A summary of these statistics can be found in Figure 4.7.

Lemma Category	Analysis Time (seconds)	Number of occurrences
Key Secrecy	0.25	8
Token Secrecy	0.25	8
1-level encryption messages	1	8
1-level encryption	1	1
2-level encryption messages	12	8
2-level encryption	1	2
Message Encryption Depth	2	8
Protocol Encryption Depth	2	1
Total time (seconds)	129	

Figure 4.7: Comparative Effort Analysis

Agreement is far less automated than secrecy, so effort analysis is of less interest. However, preliminary results indicate that we still see similar benefits in time as we do for secrecy. Most of this is due to the extension construction, since this would normally have to be done manually, and it requires a great deal of effort to construct the extension in the prover simply because of its code size. As we have discussed the secrecy proof for SET in Section 4.2.3, we will not discuss the proof of the secrecy properties for NetBill.

4.5 General Lessons

Effort analysis implies much increased efficiency with respect to both time and computational effort, especially as the number of properties increases. While it took considerable initial effort to understand both PVS and the protocol descriptions themselves, we believe that this effort would be quickly transferable to new systems, and that the overall time to prove properties would be low for two or more properties. If only a single property for a single protocol was of interest, a hand-coded system may make things easier, although then, the risk of encoding errors makes the results less reliable unless the encoding is rigorously checked for internal consistency, as we have done.

The advantages, with respect to total proof time, of breaking down the system into a large number of sub-lemmas are evident from the effort analysis shown earlier in this case study. However this technique also solves another, perhaps even more important problem. When proving the theorems without the use of supporting lemmas, it is impossible for the operator to remember the current overall state of the proof, due to its large size. PVS does attempt to alleviate this problem by providing an interactive tree structure for visualising the proof. However, this tree becomes far too slow to manipulate when the proofs reach a moderate size. Thus in larger proofs it is easy for the operator to fail to realise that they have taken the wrong path and become trapped in a never ending recursion, since they have no contextual information to draw on. It also causes the opposite problem, where the operator may believe that they are in an infinite loop, and so back up several steps to reattempt the proof, when in actual fact they were simply in a large iteration. Both of these problems are addressed by our system of breaking the proof down, and the automated strategies also lessen the mental load on the operator, allowing them to remember a larger section of the current context associated with the proof.

Having used two separate protocols in our analysis, we have confidence in the general nature of our system. Also, due to our hand analysis of two other protocols, IKE and Millicent, both of which followed a similar pattern to NetBill and SET, we believe that our overall structure is sufficiently generic

to accommodate most reasonable security protocols.

Chapter V

Conclusions and Future Work

5.1 Conclusions

We have developed a system which allows the translation of a protocol description given in a modified CAPSL/Casper notation, into a formal description for the theorem proving tool PVS. This translation generates a state transition system, and associated rules to describe the action of the protocol. It also automatically produces theorems describing the desirable security properties of the protocol, and lemmas describing subsets of these theorems, which are used to assist in the final proofs.

Our translation tool is fully automated. The input language is also simple, and although it deviates from the CAPSL format, it is sufficiently similar to the core language to be easily recognised. Thus the only section requiring in-depth knowledge and time is the interaction with the PVS protocol encoding. Our attempts to automate this fall into two broad categories, lemmas and strategies. We break down the proofs into manageable lemmas, and then attempt to automatically prove as many of these as possible using prewritten PVS/LISP strategies. Our lemma design is the basic step, and the strategies are then written to conform to this design.

5.1.1 Secrecy

For secrecy, there are two general classes of lemmas generated: key secrecy and token secrecy. For token secrecy, we are interested in those tokens which are defined as being secure in the input protocol definition. For keys, we assume that no key is revealed by a valid protocol run, although we do not make assumptions about the Spy's initial knowledge. These two classes

of lemmas are almost identical in their definitions, simply stating that no message transmits the token of interest in a non-encrypted form. This is discussed as `reveals`, in Section 3.3.4. From our overall proof structure, we see that these lemmas will provide us with the basis of our zero-level encryption system, as shown in Figure 3.17.

Dealing with 1-level encryption is more complex than for zero-level encryption. We now have to consider the case where a key is revealed in one message, to be used in another. This means we can no longer break the process down into a message based system; we have to deal with the protocol as a whole. These lemmas are approximately twice as complex, with respect to proof effort, as those relating to zero-level encryption.

After 1-level encryption we simply progress through, dealing with as many depths as required until we have reached the maximal depth for our protocol. For example the NetBill protocol contains doubly nested encryption. We thus need to go as far as two levels in these lemmas, as the third case would simply not be part of the protocol. This stopping criterion is encoded in a maximal depth lemma. This lemma states that there cannot be any combination of fields, such that three levels of encryption result, as this does not occur in the protocol. Thus any field which involves more than this level can immediately be disregarded, as it cannot be part of a conforming trace. We can again break up this encryption rule. Since encryption is confined to one message, we simply break it down on a message by message basis. Using all of these above lemmas, we have removed a large amount of user intervention from the proofs of both SET and NetBill. Our techniques now allow the user to focus on the core problems in the protocol, as these are the places where the standard strategies will fail, leaving the user in a state which causes the tool problems. As these problems are overcome the strategies can be updated to reflect these advances. However it is important to maintain the protocol independent nature of the strategies, and thus not all aspects of a specific proof may be generalisable enough to be applicable.

5.1.2 Agreement

We have not been as successful with agreement as we have been with secrecy, as agreement is a type of liveness property, which are generally harder to prove. However, we have identified a technique for the proof, and given a supporting structure for this technique. Agreement does have one advantage over secrecy, as far as simplicity of proof, since it only requires the existence of a solution, and an exhaustive search is not required Therefore we do not have to delve into the structure and content of every message in the system, only those which we believe are of interest.

Our agreement technique is as follows. There are two general classes into which an agreement property can fall: either one host knows the given token initially, and the other then learns it through the course of the protocol, or neither host knows it, and both must learn it. The first of these two cases is the simpler. In this case the analytical effort is undertaken by only one host, as the initial knowledge predicates will trivially satisfy the other host. Our translator now generates two sub-lemmas, one for each host. Only one of these is of interest to us, that for the host who must learn the token. The majority of the effort of the proof is in this sub-lemma, which was discussed in detail in Section 3.6. Once this sub-lemma is proven, the main theorem is simply broken into two cases, one for each host. The case for the host who learned it by initial knowledge is trivially proven, and the case for the other host is proven via instantiating the above sub-lemma. It should be noted that although we generate two sub-lemmas, only one will be used.

The other agreement scenario, in which neither host knows the given token in initial knowledge, is slightly more complex. Here we again take one of the sub-lemmas generated, except now the choice of sub-lemma is more difficult. The operator must inspect the protocol and determine which of the two extensions generated by the heuristic is valid for *both* hosts to learn. If neither are valid, one must be constructed by hand; although this construction is simple to do by modifying one of those auto-generated. Once the correct extension has been constructed, it is instantiated into one of the agreement lemmas, which is then proved in a similar manner to the case discussed above. Once this lemma is completed it can then be instantiated

into the final theorem, thus solving half of the agreement. We now have the given extension in place, and only one host remaining, thus a similar proof as for the sub-lemma is carried out. Although this does not break as cleanly into subsections as the other case, it is still easy to use within the environment.

The proving of the sub-lemmas is quite similar to that for secrecy, except in reverse. We are now trying to find a message in which a given host can learn a token. This is almost always within an encrypted message for any reasonable protocol, as agreement tokens are generally secure. There are generally multiple messages of interest for a given host and given token. For example, the token may be sent in one message, but only in its encrypted form, and the key for that may be negotiated via a public/private key handshake. In this case there would be three messages of interest, each with varying levels of encryption. We would then walk through the expansion of `knows`, and use these messages to show how the token is learnt.

5.1.3 Strategies

The strategies we have developed are straightforward; we have not used many of the PVS's advanced techniques. However, they have been quite successful at simplifying the act of proving those lemmas for which they have been written. The main drawback when developing these strategies is the difficulty in keeping them generic. It is very easy to put in protocol specific constants. For example, in the encryption depth strategy, depending on the depth of the given protocol, a certain equation would be placed on a different line. Our original strategy would then fail here, rewriting the wrong line. These problems are easily removed once they have been identified, but are often difficult to determine initially.

5.1.4 General Comments

At the moment there is a single strategy definition for all protocols, as we have not yet found a need for protocol dependent strategies. However, it would be a trivial task to alter this to use a generalised file, as well as a protocol dependent file, which could also be created from the translation

tool, or coded by hand if that were desirable. The use of parameterised strategies has allowed us to retain generalisation so far.

Our initial goal, as we discussed in Chapter 1, was to develop a system for automating the encoding and proof of security properties. We believe that we have succeeded in these goals, our encoding step being completely automated, and though the proof stage remains partially manual, we believe that this is a result of the undecidability of several portions of the proof, and a completely automated system would be impossible for all classes of protocol, due to the necessarily unbounded nature of our model. Our case study analysis has demonstrated that the effort involved in these proofs can be drastically reduced through the use of our system.

5.2 Future Work

5.2.1 Analyse Additional Protocols

While we have attempted to remain general in our encoding, primarily by investigating both NetBill and SET simultaneously, we realise that our encoding may be biased towards certain types of protocol design. We would thus like to describe and verify various other protocols to ensure that our system behaves well under a variety of conditions. During the course of the analysis the protocol Millicent [Man95] was proposed as an alternative protocol to examine, rather than NetBill. However its integration with the HTTP [BLFF96] protocol makes it difficult to extract the core protocol information. We would also be interested in protocols which involve internal calculations, see Section 5.2.3.

5.2.2 General Spy Properties

In our input language definition, we have added the ability to describe certain properties of the Spy. However, only message interception has been implemented. The other two properties described, message destruction and message creation, allow for a much greater range of protocol analysis, as they force protocols to be tamper proof, and force hosts to be more involved in the verification of alternate hosts. Since the current design separates Spy prop-

erties from protocol description, it would not be difficult to have a library of PVS definitions for various logics, which could be enabled and disabled via the translation tool.

5.2.3 Computational Rules

Computational rules are again implemented in the input language, although not fully carried through to the protocol encoding. We allow each message to define any number of calculations to be performed, as well as any number of logical checks on any of the variables involved in a protocol. Thus for protocols such as IKE [HC98], where the shared key is calculated by each host during the course of the communication, we can encode this information, and allow for a more in-depth analysis. This is particularly important when secure information can be broken without complete knowledge of the keys involved, but through some known mathematical analysis of a subset of the required data. The current implementation of the tool provides the infrastructure for this analysis, but none of the higher level code.

5.2.4 Anonymous Hosts

In the Needham-Schroeder [NS78] protocol the initial message is communicated from an anonymous host, such that the receiver cannot infer any information about the sender. We can currently emulate a similar behaviour by adding an additional host from whom these anonymous messages can be sent, but this could cause problems with unexpected side effects, i.e.: if there are two anonymous messages, are they from the same host? It would be more elegant if this were handled in an automated manner to remove the chance of an input error by the user.

5.2.5 Trace Run Identifiers

In the current encoding, each variable is associated with a host, to allow variables to change values between runs. This feature does, however, cause some semantic issues when trying to determine the value of a variable, as it may not be associated with the correct host. An alternative to this encoding

would be to attach a unique session number to each new protocol initialisation, and then bind the variables using this session number. This would add a small amount of complexity to the encoding, but would not interfere too severely. The changes would be mostly superficial, so conceptually most of the encoding would remain the same. However from an implementation and strategy point of view, this would require changes to most sections of the code. This requirement for large changes made this extension infeasible initially, especially due to its dubious benefits, as it only really affects readability and ease of understanding, rather than the core abilities of the encoding.

5.2.6 Initial Knowledge Encoding

As we have divided our encoding into protocol dependent and protocol independent subsystems, when dealing with the `knows` predicate it is important to instantiate the initial knowledge axioms as appropriate. This has the slight drawback that the operator must remember to do this. If this is forgotten, it does not cause any theorems to be proved which would otherwise not be, but it does mean that some theorems remain unproven, where initial knowledge could be used. An alternative encoding would be to insert initial knowledge into the `knows` predicate. However this would needlessly duplicate the facts, as `knows` is used quite frequently. This would also cause problems with finding the correct round bindings, since if we have a multi-round run of a protocol, we cannot be sure which round of hosts we are interested in at the time we must instantiate `knows`. This would also complicate the encoding, as we could no longer distinguish the independent sections as easily. We have not currently found a suitable compromise between these two extremes.

5.2.7 Fuller Analysis of Non-standard Protocols

All of our current analysis has been towards protocols which fit a certain model, such as not having self-referential processes. We would like to also analyse protocols which contain slight differences from this norm to ensure that our lemma and strategy system is capable of scaling to these systems. We do not anticipate these having any impact on our encoding model, since it

is already a very general system, but our strategies, in particular, often expect a certain flow from the system. We expect a steady progression through the system, with recursion being handled by our encoding, rather than by the protocol. Here we are more interested in stress testing our encoding system. Both of our case studies were quite standard security protocols, and while this does provide well-rounded results, it also limits the extreme scenarios which we have analysed.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, principles, techniques and tools.* Addison-Wesley, 1986.

[BAN89] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. In *Proceedings of the Royal Society of London A*, pages 233–271, 1989.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, May 1996. `http://www.rfc-editor.org/rfc/rfc1945.txt`.

[BMPT00] G. Bella, F. Massacci, L. C. Paulson, and P. Tramontano. Formal verification of cardholder registration in SET. In *ESORICS*, volume 1895 of *LNCS*, pages 159–174, 2000.

[Bur95] J. H. Burrows. FIPS 180-1: Secure Hash Standard, April 1995. U.S. Department of Commerce and National Institute of Standards and Technology, `http://www.itl.nist.gov/fipspubs/fip180-1.htm`.

[COR+95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, 1995.

[CTS95] B. Cox, J. D. Tygar, and M. Sirbu. NetBill Security and Transaction Protocol. *First USENIX Workshop on Electronic Commerce*, pages 77–88, July 1995. `http://www.ini.cmu.edu/NETBILL/pubs/Usenix.html`.

[DA99] T. Dierks and C. Allen. The TLS Protocol. RFC 2246, January 1999. `http://www.rfc-editor.org/rfc/rfc2246.txt`.

[DM99] G. Denker and J. K. Millen. CAPSL and SIL Language Design. Technical Report SRI-CSL-99-02, SRI International, February 1999.

[DS95] C. Donnelly and R. Stallman. *Bison – The YACC-compatible Parser Generator.* Free Software Foundation, November 1995. ISBN 1-882114-45-0.

[HC98] D. Harkins and D. Carrel. The internet key exchange. RFC 2409, November 1998. `http://www.rfc-editor.org/rfc/rfc2409.txt`.

[Hoa85] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985. ISBN 0-13-153271-8.

[Low95] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science.* Springer Verlag, March 1996.

[Low98] G. Lowe. Casper: A compiler for the analysis of security. *Journal of Computer Security*, 6(1):53–84, 1998.

[Man95] M. S. Manasse. The Millicent protocols for electronic commerce. In *Proceedings of the First USENIX Workshop of Electronic Commerce*, Berkeley, CA, USA, 1995. USENIX Assoc.

[MCJ97] W. Marrero, E. Clarke, and S. Jha. Model Checking for Security Protocols. Technical Report CMU-CS-97-139, Carnegie Mellon University, 1997.

[Mea96] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[Mil90] R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 1201–1242, 1990.

[Mil99] J. K. Millen. A necessarily parallel attack. In N. Heintze and E. Clarke, editors, *Workshop on Formal Methods and Security Protocols, Part of the Federated Logic Conference*, Trento, Italy, July 1999.

[MR00] J. K. Millen and H. Ruess. Protocol-independent secrecy. In *RSP: 21st IEEE Computer Society Symposium on Research in Security and Privacy*, pages 110–119, May 2000.

[MS98] C. Meadows and P. Syverson. A Formal Specification of Requirements for Payment Transactions in the SET Protocol. *Preproceedings of Financial Cryptography*, February 1998.

[NS78] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[Pau99] L. Paulson. Inductive analysis of the internet protocols TLS. In *ACM Transactions on Computer and System Security*, volume 2 of *3*, pages 332–351, 1999.

[Pau01] L. Paulson. Introduction to Isabelle. Technical report, Computing Laboratory, University of Cambridge, Februrary 2001. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle99-2/doc/int%ro.pdf`.

[RK01a] A. Renaud and P. Krishnan. An Environment for Specifying and Verifying Security Properties. To appear in *Australian Software Engineering Conference*, August 2001.

[RK01b] A. Renaud and P. Krishnan. Automatic verification of agreement and secrecy. In *New Zealand Computer Science Research Student Conference*, volume TR-COSC 02/01, pages 48–55, April 2001. To appear in the New Zealand Journal of Computing, 2001.

[Ros94] A. W. Roscoe. Model-checking CSP. *A Classical Mind, Essays in Honour of C. A. R. Hoare*, 1994. Prentice-Hall.

[Ste84] G. L. Steele. *Common LISP: The Language.* Digital Press, 1984. ISBN 0-932376-41-X.

[TY98] C. L. Tocq and S. Young. SET Comparative Performance Analysis. Technical report, Gartner Group, November 2 1998.

[VIS97] VISA International. *SET Secure Electronic Transaction Specification*, 1.0 edition, May 31 1997. `http://www.visa.com/nt/ecomm/security/set.html`.